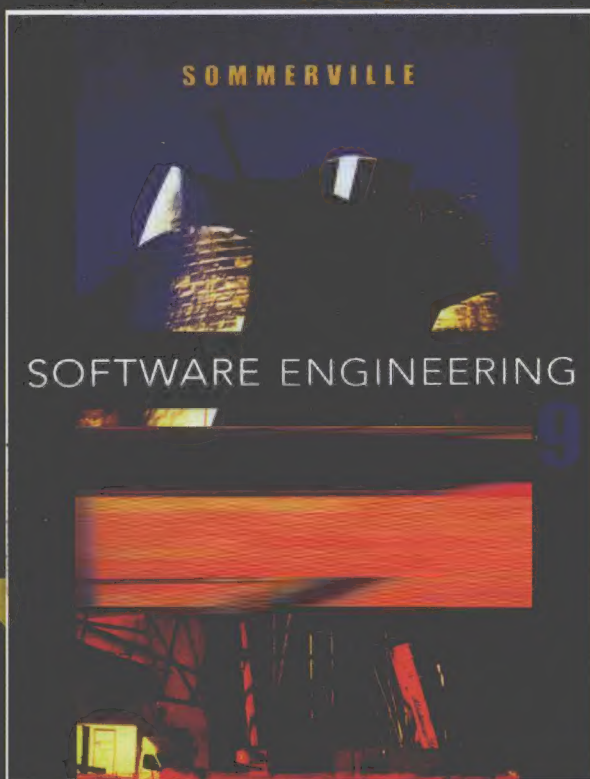


软件工程

(英) Ian Sommerville 著 程 成 等译
圣安德鲁斯大学 北京理工大学

Software Engineering
Ninth Edition



软件工程 (原书第9版)

Software Engineering Ninth Edition

本书是系统介绍软件工程理论的经典教材,自1982年初版以来,随着软件工程学科的发展不断更新,影响了一代又一代软件工程人才,对学科本身也产生了积极影响。全书共四个部分,完整讨论了软件工程各个阶段的内容,是软件工程和系统工程专业本科和研究生的优秀教材,也是软件工程师必备的参考书籍。

本书特点

- 涵盖了对所有开发过程都很基础的重要主题,包括软件工程理论与实践的最新进展。
- 将本书第8版中的八篇内容重构为四个部分,使教师讲授软件工程课程更加容易。
- 每一章都有30%~40%的更新,增加了敏捷软件开发和嵌入式系统等新章,补充了模型驱动工程、开源开发、测试驱动开发、可依赖系统体系结构、静态分析和模型检查、COTS复用、服务作为软件以及敏捷规划等新内容。
- 着重讨论了开发可靠的分布式系统的相关主题以及敏捷方法和软件复用。
- 反映敏捷方法先进性的同时,不忘强调传统的计划驱动软件工程的作用,阐述了两者的结合构建优秀软件系统的重要性。
- 以一个新的病人记录系统案例研究贯穿始终,系统、完整地讲解软件工程的各个方面。
- 将本书设计为“印刷/Web”相结合的方式,核心信息采用印刷版本,教辅材料及先前版本中的一些章节放在Web上,为读者提供丰富翔实的信息。



英文版·第9版
即将出版



PEARSON

www.pearsonhighered.com

客服热线: (010) 88378991, 88361066
购书热线: (010) 68326294, 88379649, 68995259
投稿热线: (010) 88379604
读者信箱: hzjsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书: www.china-pub.com

封面设计: 李易 林杉



上架指导: 计算机·软件工程

ISBN 978-7-111-33498-9



9 787111 334989

定价: 75.00元

计 算 机 科 学 丛 书

原书第9版

软件工程

(英) **Ian Sommerville** 著 程 成 等译

圣安德鲁斯大学

北京理工大学

Software Engineering

Ninth Edition

SOFTWARE ENGINEERING



机械工业出版社
China Machine Press

本书是系统介绍软件工程理论的经典教材,自1982年初版以来,随着软件工程学科不断发展,不断更新版本,影响了一代又一代软件工程人才,对学科本身也产生了重大影响。本版保留了上一版中的软件工程的基本材料,但对各章都进行了修改和更新,并增加了很多有关其他主题的新材料。

本书包含四个部分:第一部分是对软件工程的一般性介绍,包括软件工程过程和敏捷开发,以及面向对象的设计和模式的使用;第二部分介绍可依赖性和信息安全性问题;第三部分介绍高级软件工程;第四部分介绍软件管理,重点介绍技术管理问题。

本书适合作为软件和系统工程专业本科生或研究生教材,同时也是软件工程师难得的优秀参考书籍。

Simplified Chinese edition copyright © 2011 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Software Engineering, Ninth Edition* (ISBN 978-0-13-703515-1) by Randal E. Bryant and David R. O'Hallaron, Copyright © 2011, 2006, 2005, 2001, 1996.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

本书封面贴有 Pearson Education (培生教育出版集团) 激光防伪标签,无标签者不得销售。

封底无防伪标均为盗版

版权所有,侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号: 图字: 01-2010-2459

图书在版编目(CIP)数据

软件工程(原书第9版)/(英)萨默维尔(Sommerville, I.)著;程成等译. —北京:机械工业出版社, 2011. 4

(计算机科学丛书)

书名原文: *Software Engineering, Ninth Edition*

ISBN 978-7-111-33498-9

I. 软… II. ①萨… ②程… III. 软件工程—教材 IV. TP311.5

中国版本图书馆 CIP 数据核字 (2011) 第 026678 号

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑: 刘立卿

北京诚信伟业印刷有限公司印刷

2011 年 5 月第 1 版第 1 次印刷

185mm × 260mm · 30.25 印张

标准书号: ISBN 978-7-111-33498-9

定价: 75.00 元

凡购本书,如有缺页、倒页、脱页,由本社发行部调换

客服热线: (010) 88378991; 88361066

购书热线: (010) 68326294; 88379649; 68995259

投稿热线: (010) 88379604

读者信箱: hzsj@hzbook.com

文艺复兴以降,源远流长的科学精神和逐步形成的学术规范,使西方国家在自然科学的各个领域中取得了垄断性的优势;也正是这样的传统,使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中,美国的产业界与教育界越来越紧密地结合,计算机学科中的许多泰山北斗同时身处科研和教学的最前线,由此而产生的经典科学著作,不仅擘划了研究的范畴,还揭示了学术的源变,既遵循学术规范,又自有学者个性,其价值并不会因年月的流逝而减退。

近年,在全球信息化大潮的推动下,我国的计算机产业发展迅猛,对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇,也是挑战;而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下,美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此,引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用,也是与世界接轨、建设真正世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始,我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力,我们与Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage等世界著名出版公司建立了良好的合作关系,从他们现有的数百种教材中甄选出Andrew S. Tanenbaum, Bjarne Stroustrup, Brian W. Kernighan, Dennis Ritchie, Jim Gray, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, Abraham Silberschatz, William Stallings, Donald E. Knuth, John L. Hennessy, Larry L. Peterson等大师名家的一批经典作品,以“计算机科学丛书”为总称出版,供读者学习、研究及珍藏。大理石纹理的封面,也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助,国内的专家不仅提供了中肯的选题指导,还不辞劳苦地担任了翻译和审校的工作;而原书的作者也相当关注其作品在中国的传播,有的还专程为其书的中译本作序。迄今,“计算机科学丛书”已经出版了近两百个品种,这些书籍在读者中树立了良好的口碑,并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑,这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化,教育界对国外计算机教材的需求和应用都将步入一个新的阶段,我们的目标是尽善尽美,而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正,我们的联系方式如下:

华章网站: www.hzbook.com

电子邮件: hzjsj@hzbook.com

联系电话: (010) 88379604

联系地址: 北京市西城区百万庄南街1号

邮政编码: 100037



华章科技图书出版中心

又一轮桐庭落叶，兰畦浮香，银蟾新满，我们再次拿出了本书的中文译稿。此时的心情轻松愉悦。往日的辛劳没有让我们感到任何痛苦，近 800 页原著的每一页犹如一杯杯醇香咖啡，令人回味无穷。自从本书第 6 版中文译著由我们翻译出版以来，一直得到众多读者的帮助和厚爱，所以对这次第 9 版的翻译我们更加珍惜，倍加努力。如果要问我们的感想，那就是通过此书对社会尽一点绵薄之力，让同胞一同分享该书的快乐了。

本书是一部阐述软件工程理论和技术的著作，几乎涵盖了软件工程的所有方面，同时，就像本书作者所声明的那样，由于作者对需求分析和实时系统的特殊偏好，使得本书极具特点。本书的第一个特点是，通篇阐述主要以要求极高的一类系统为实例，使得本书的理论和方法阐述非常容易理解。同时，译者认为，由于要求极高的一类系统具有一般软件系统的几乎所有特性，而且具备一般软件不必要的很多特征，因此，掌握本书精髓，对于一般软件系统的研究和开发无疑会深得要领且更加得心应手。本书的第二个特点是形式化描述内容较多，这一方面与要求极高的一类系统设计有关，另一方面也多少反映出一些欧洲学者的研究之风。作者将形式化描述结合具体应用实例阐述得深入浅出，是我们学习和掌握该方法的极好教材。本书的第三个特点就是关于软件进化理论和方法的系统介绍。译者也读过其他的一些著作，但比较起来在软件进化理论和方法的阐述上本书当属最透彻的一本了。有过软件进化相关工程经历的人经常会有困惑之处，本书在这方面的高屋建瓴般的阐述无疑将对同行们起到答疑解惑的作用。本书的第四个特点是，通过对本书第 8 版的改造，从多个角度集中介绍了在快速软件开发等方面的技术，以反映当前软件领域的最新动向。以上这些只是我们的一点粗浅认识，相信读过本书之后，读者会有更多更深的领悟。不过本书内容之系统翔实，阐述之精辟透彻，引文和材料之丰富，确实让我们叹为观止。

本书主要由程成翻译。参与本书翻译和审校等相关工作的还有北京理工大学计算机学院的研究生蔡雪琴、董雪梅、姜儒、刘晓峰、周小飞、马鹏飞、高彦明、邵霞、刘佳，在此对他们的工作热情和认真态度以及对本书翻译的大力帮助表示感谢。

鉴于译者自身的知识局限及时间仓促，译稿中难免有错误和遗漏之处，谨向原书作者表示歉意，并欢迎广大读者批评指正。

程成
北京理工大学

2009年夏，当我写本书最后一章的时候，软件工程正好有40年的历史了。“软件工程”这个名字是在1969年的NATO大会上讨论软件开发问题时提出来的。所讨论的开发问题是，大型软件系统总是延期，不能交付用户期待的功能，成本超出预期，软件不可靠。本人并没有参加那次会议，但是一年之后，我写了我的第一个程序并开始了我的软件职业生涯。

在我的整个职业生涯中软件工程的进展是不同凡响的。我们的社会如果没有大型专业软件系统就根本不能运转。对于构建业务系统，这里有一串技术习语，J2EE、.NET、SaaS、SAP、BPEL4WS、SOAP、CBSE等，它们支持大型企业应用的开发和部署。国家公用事业和基础设施（能源、通信以及运输）都依赖于复杂而可靠的计算机系统。软件让我们能够探索空间，创建万维网这个人类历史上最为强大的信息系统。人类正面临着新的挑战——气候变化和极端天气，自然资源的减少，需要为更多的人口提供食物和住房，国际恐怖主义的威胁，还有需要为老年人提供更加满意的生活。我们需要新技术来帮助我们解决这些问题。而且可以肯定，软件将在这些技术中扮演核心角色。

因此，软件工程是人类未来十分重要的技术。我们必须不断地教育软件工程师和发展这门学科，使我们能构造出更加复杂的软件系统。当然，现在我们的软件项目还有很多问题，软件仍然在延期，成本超过预算。但是，我们不应该让这些问题遮挡住视线，应该看到在软件工程上取得的真正的成功，应该看到我们已经研究出了了不起的软件工程方法和技术。

软件工程现在是一个非常大的领域，任何一本书都不可能覆盖它的所有主题。因此，我的重点放在对所有开发过程都很基础的重要主题上，以及放在关于开发可靠的分布式系统的一些主题上。对敏捷方法和软件复用给予了更多的关注。我坚信敏捷方法会有它们的位置，但是“传统”的计划驱动软件工程也肯定不会消失。我们需要将这些方法中的最好的东西结合起来以构建更好的软件系统。

任何一本书都不可避免地反映作者的观点和倾向。肯定会有某些读者不赞成我的观点和我所选择的材料。这种不同意见是学科多样性的一种健康反映，对学科的演化也是很有好处的。尽管如此，我还是希望所有的软件工程师和学习软件工程的学生能从本书中发现自己感兴趣的东西。

与 Web 的整合

在 Web 上关于软件工程有非常大量的信息可用，因此很多人会问是否有必要写这样一本教科书。然而，网络上的可用信息是很不完整的，质量也是参差不齐的，有的信息呈现的形式很差，很难找到你想要的内容。所以，我相信教科书在学习上仍然有不可替代的作用。它们会像路标一样指引你奔向主题，可以将方法和技术的内容很好地组织在一起方便你阅读。它们也会为你提供一个深入探索研究文献和网上可用信息的起点。

我坚信，只有当教科书与网络上的材料结合在一起并能众多可用信息增加价值的时候，它们才是有前途的。本书因而设计成混合的“印刷/Web”这样的文本形式。核心信息用印刷版本，教辅材料放在 Web 上。几乎所有的章都包含特别的“Web 小节”，成为该章内容的补充。另外还有4个“Web 章”，这4章内容没有在本书印刷版中出现。

与本书相关的 Web 站点是：

<http://www.SoftwareEngineering-9.com>

本书的网站有 4 个主要部分：

1. **Web 小节 (Web sections)** 一些额外的小节，是为每一章所添加的部分内容。这些 Web 小节是通过每章当中用方框括起的内容指引的。

2. **Web 章 (Web chapters)** 这 4 个 Web 章覆盖了形式化方法、交互设计、文档化以及应用体系结构。今后，我会在本书的升级版本中添加其他新的主题到“Web 章”中。

3. **教师材料** 此部分的材料是为那些讲授软件工程课程的教师准备的。参见前言后面的“支持材料”。

4. **案例研究** 提供关于本书中使用的案例研究的附加信息（胰岛素泵、心理健康护理系统、野外气象站系统），也包括另外更深入的案例信息，如阿丽亚娜 5 号火箭发射失败。

除此之外，本书还有到其他网站的链接，包括关于软件工程、进一步阅读、博客以及通讯简报等有价值材料的网站。

我很欢迎读者提出关于本书及其网站的建设性意见和建议。读者可以通过 ian@SoftwareEngineering-9.com 与我联系。请在主题中注明 [SE9] 字样，否则，我的邮件过滤器会拒绝你的邮件因而得不到我的回信。我本人没有时间帮助学生解答他们的课后作业，所以请不要问类似问题。

读者对象

本书主要面向各大学和学院正在学习软件和系统工程的初高级课程的学生。工业界的软件工程师也会发现它是一本很好的读物，能帮助他们更新在软件复用、体系结构设计、可依赖性和信息安全性以及过程改善等方面的知识。我假设读者都完成了初级的编程课程学习并了解编程方面的术语。

对先前版本的变更

这一版保留了上一版中的软件工程的基本材料，但是我还是修改和更新了所有章节的内容，并增加了很多有关其他主题的新材料。其中最重大的改变有：

1. 从单纯的印刷版转变到混合的“印刷/Web”版，将 Web 材料与印刷材料紧密结合在一起。这就允许我减少本书章的数量，且能更加专注于每一章的核心内容。

2. 结构变化很大使得教师讲授软件工程课程会更加容易。本书现在有 4 个部分而不是 8 个部分，每个部分可以单独使用或者是结合其他部分作为软件工程课程的基本内容。这 4 个部分分别是软件工程师导论、可依赖性和信息安全性、高级软件工程和软件管理。

3. 将来自先前版本的多个主题压缩到一章中阐述，并与其他材料一起放在 Web 上。

4. 额外的 Web 章，那些出现在先前版本中而没有包含在本书中的各章，现在都放在 Web 上了。

5. 更新和修改了每一章。我估计有 30% ~ 40% 的内容都经过全面彻底的重写。

6. 增加了关于敏捷软件开发和嵌入式系统的一些章。

7. 除了这些新的章节外，在以下方面也增添了新的内容：模型驱动工程、开源开发、测试驱动开发、Reason 的瑞士奶酪模型、可依赖系统体系结构、静态分析和模型检测、COTS 复用、作为服务的软件以及敏捷规划。

8. 关于病人记录系统的新案例研究，是在很多章中都会用到的关于接受心理健康问题治疗的病人的信息系统。

用作教学

针对 3 种不同类型的软件工程课程，我对本书进行了如下设计：

1. **软件工程一般性介绍课程** 本书第一部分就是设计来明确支持一个学期的介绍软件工程基本内容的课程。

2. **特别软件工程主题的介绍课程或过渡课程** 通过使用第二～四部分的内容，可以创建一系列更高级课程。例如，我采用第二部分的各章加上关于质量管理和配置管理的两章讲授要求极高的系统工程课程。

3. **关于特别软件工程主题的更高级的课程** 在此情况下，本书的各章可以构成课程的基础，然后辅之以更多的阅读以便进一步探索某个主题。例如，关于软件复用的课程就可以基于第 16～19 章的内容。

支持材料

本书带有大量支持材料帮助教师采用此书作为软件工程教材授课。这些材料包括：

- 本书所有章的幻灯片。
- 幻灯片中的图和表。
- 教师指导，包括一些关于如何在不同的课程中利用本书的建议，并解释了在此版和先前版本中各个章之间的关系。
- 关于书中案例研究的进一步资料。
- 会在软件工程课程中用到的其他案例研究。
- 有关系统工程的其他幻灯片。
- 4 个 Web 章，涵盖了形式化方法、交互设计、应用体系结构以及文档化。
- 部分章后练习的答案。

所有上述材料都是免费提供给本书读者的，可以从本书的网站 www.hzbook.com 下载。

致谢

在过去的几年时间里有很多人对本书的改进做出了贡献，我想在此感谢所有的人（审阅人、学生、读者），他们对本书先前版本给出了很多评论，对本书的更新给出了建设性的意见。

特别要感谢我的家人（Anne、Ali 和 Jane），没有她们的帮助和支持，我是不可能完成本书的写作的。尤其是我的女儿 Jane，我要特别谢谢她。她利用自己校对和编辑方面的天赋，在通读全书的过程中给了我巨大的帮助，帮助我发现和校正了大量的拼写错误和语法错误。

Ian Sommerville

2009 年 10 月

目 录

Software Engineering, 9E

出版者的话

译者序

前言

第一部分 软件工程导论

第 1 章 概述	2
1.1 专业化软件开发	3
1.1.1 软件工程	4
1.1.2 软件工程的多样性	6
1.1.3 软件工程和 Web	7
1.2 软件工程人员的职业道德	8
1.3 案例研究	10
1.3.1 胰岛素泵控制系统	10
1.3.2 用于心理健康治疗的患者 信息系统	12
1.3.3 野外气象站	13
要点	14
进一步阅读材料	14
练习	15
参考书目	15
第 2 章 软件过程	16
2.1 软件过程模型	17
2.1.1 瀑布模型	17
2.1.2 增量式开发	19
2.1.3 面向复用的软件工程	20
2.2 过程活动	21
2.2.1 软件描述	21
2.2.2 软件设计和实现	23
2.2.3 软件有效性验证	25
2.2.4 软件进化	26
2.3 应对变更	26
2.3.1 原型构造	27
2.3.2 增量式交付	28
2.3.3 Boehm 的螺旋模型	29
2.4 Rational 统一过程	30

要点	32
进一步阅读材料	33
练习	33
参考书目	34
第 3 章 敏捷软件开发	35
3.1 敏捷方法	36
3.2 计划驱动开发和敏捷开发	38
3.3 极限编程	40
3.3.1 极限编程中的测试	42
3.3.2 结对编程	44
3.4 敏捷项目管理	45
3.5 可扩展的敏捷方法	46
要点	48
进一步阅读材料	48
练习	48
参考书目	49
第 4 章 需求工程	51
4.1 功能需求和非功能需求	52
4.1.1 功能需求	52
4.1.2 非功能需求	53
4.2 软件需求文档	56
4.3 需求描述	58
4.3.1 自然语言描述	58
4.3.2 结构化描述	59
4.4 需求工程过程	61
4.5 需求导出和分析	62
4.5.1 需求发现	63
4.5.2 采访	64
4.5.3 脚本	65
4.5.4 用例	65
4.5.5 深入实际	67
4.6 需求有效性验证	68
4.7 需求管理	69
4.7.1 需求管理规划	70
4.7.2 需求变更管理	71
要点	71

进一步阅读材料	72	7.1.3 对象类识别	116
练习	72	7.1.4 设计模型	117
参考书目	73	7.1.5 接口描述	120
第5章 系统建模	74	7.2 设计模式	120
5.1 上下文模型	75	7.3 实现问题	123
5.2 交互模型	77	7.3.1 复用	123
5.2.1 用例建模	77	7.3.2 配置管理	124
5.2.2 时序图	78	7.3.3 宿主机 - 目标机开发	124
5.3 结构模型	80	7.4 开源开发	126
5.3.1 类图	81	要点	128
5.3.2 泛化	82	进一步阅读材料	128
5.3.3 聚合	83	练习	128
5.4 行为模型	83	参考书目	129
5.4.1 数据驱动的建模	84	第8章 软件测试	131
5.4.2 事件驱动模型	84	8.1 开发测试	134
5.5 模型驱动工程	87	8.1.1 单元测试	135
5.5.1 模型驱动体系结构	88	8.1.2 选择单元测试案例	136
5.5.2 可执行 UML	89	8.1.3 组件测试	138
要点	90	8.1.4 系统测试	139
进一步阅读材料	90	8.2 测试驱动开发	141
练习	90	8.3 发布测试	143
参考书目	91	8.3.1 基于需求的测试	143
第6章 体系结构设计	93	8.3.2 情景测试	144
6.1 体系结构设计决策	95	8.3.3 性能测试	144
6.2 体系结构视图	96	8.4 用户测试	145
6.3 体系结构模式	97	要点	147
6.3.1 分层体系结构	99	进一步阅读材料	147
6.3.2 容器体系结构	100	练习	148
6.3.3 客户机 - 服务器体系结构	101	参考书目	148
6.3.4 管道和过滤器体系结构	102	第9章 软件进化	149
6.4 应用体系结构	103	9.1 进化过程	150
6.4.1 事务处理系统	105	9.2 程序进化的动态特性	152
6.4.2 信息系统	105	9.3 软件维护	154
6.4.3 语言处理系统	107	9.3.1 维护预测	156
要点	108	9.3.2 软件再工程	158
进一步阅读材料	109	9.3.3 通过重构进行预防性维护	159
练习	109	9.4 遗留系统管理	160
参考书目	110	要点	163
第7章 设计与实现	112	进一步阅读材料	163
7.1 利用 UML 进行面向对象设计	113	练习	164
7.1.1 系统上下文与交互	113	参考书目	164
7.1.2 体系结构的设计	115		

第二部分 可依赖性和信息安全性

第 10 章 社会技术系统	168	参考书目	213
10.1 复杂系统	169	第 13 章 可依赖性工程	215
10.1.1 系统总体特性	171	13.1 冗余性和多样性	216
10.1.2 系统非确定性	172	13.2 可依赖的过程	217
10.1.3 成功标准	173	13.3 可依赖的系统体系结构	219
10.2 系统工程	173	13.3.1 保护性系统	220
10.3 系统采购	174	13.3.2 自监控系统体系结构	220
10.4 系统开发	176	13.3.3 N-版本编程	222
10.5 系统运行	178	13.3.4 软件多样性	223
10.5.1 人为错误	179	13.4 可依赖的编程	224
10.5.2 系统进化	180	要点	228
要点	181	进一步阅读材料	229
进一步阅读材料	181	练习	229
练习	181	参考书目	229
参考书目	182	第 14 章 信息安全工程	231
第 11 章 可依赖性与信息安全性	183	14.1 信息安全风险管理	233
11.1 可依赖性特征	184	14.1.1 生存期风险评估	234
11.2 可用性和可靠性	186	14.1.2 运行风险评估	236
11.3 安全性	189	14.2 面向信息安全的设计	236
11.4 信息安全性	191	14.2.1 体系结构设计	237
要点	192	14.2.2 设计准则	239
进一步阅读材料	193	14.2.3 部署设计	243
练习	193	14.3 系统生存能力	244
参考书目	194	要点	246
第 12 章 可依赖性与信息安全性描述	195	进一步阅读材料	246
12.1 风险驱动的需求描述	195	练习	246
12.2 安全性描述	197	参考书目	247
12.2.1 危险识别	197	第 15 章 可依赖性与信息安全保证	248
12.2.2 危险评估	198	15.1 静态分析	249
12.2.3 危险分析	199	15.1.1 检验和形式化方法	249
12.2.4 风险降低	201	15.1.2 模型检测	250
12.3 可靠性描述	202	15.1.3 自动静态分析	251
12.3.1 可靠性度量	203	15.2 可靠性测试	253
12.3.2 非功能性的可靠性需求	204	15.3 信息安全性测试	255
12.3.3 功能可靠性描述	206	15.4 过程保证	256
12.4 信息安全性描述	207	15.5 安全性和可依赖性案例	259
12.5 形式化描述	210	15.5.1 结构化论证	260
要点	212	15.5.2 结构化的安全性论证	261
进一步阅读材料	212	要点	264
练习	212	进一步阅读材料	265
		练习	265
		参考书目	266

第三部分 高级软件工程

第 16 章 软件复用	270
16.1 复用概览	272
16.2 应用框架	273
16.3 软件产品线	276
16.4 COTS 产品的复用	279
16.4.1 COTS 解决方案系统	280
16.4.2 COTS 集成系统	282
要点	284
进一步阅读材料	284
练习	285
参考书目	285
第 17 章 基于组件的软件工程	287
17.1 组件和组件模型	288
17.2 CBSE 过程	292
17.2.1 面向复用的 CBSE	293
17.2.2 基于复用的 CBSE	294
17.3 组件合成	296
要点	301
进一步阅读材料	301
练习	301
参考书目	302
第 18 章 分布式软件工程	303
18.1 分布式系统的问题	304
18.1.1 交互模型	306
18.1.2 中间件	307
18.2 客户机-服务器计算	308
18.3 分布式系统的体系结构模式	309
18.3.1 主从体系结构	310
18.3.2 两层客户机-服务器结构	310
18.3.3 多层客户机-服务器结构	312
18.3.4 分布式组件体系结构	313
18.3.5 对等体系结构	315
18.4 软件作为服务	317
要点	319
进一步阅读材料	320
练习	320
参考书目	321
第 19 章 面向服务的体系结构	322
19.1 服务作为可复用的组件	325
19.2 服务工程	328

19.2.1 可选服务的识别	328
19.2.2 服务接口设计	330
19.2.3 服务实现和部署	332
19.2.4 遗留系统服务	332
19.3 使用服务的软件开发	333
19.3.1 workflow 设计和实现	335
19.3.2 服务测试	337
要点	338
进一步阅读材料	338
练习	339
参考书目	339
第 20 章 嵌入式软件	341
20.1 嵌入式系统设计	342
20.1.1 实时系统建模	345
20.1.2 实时编程	346
20.2 体系结构模式	347
20.2.1 观察和反应	347
20.2.2 环境控制	349
20.2.3 处理管道	350
20.3 时序分析	351
20.4 实时操作系统	354
要点	356
进一步阅读材料	356
练习	356
参考书目	357
第 21 章 面向方面的软件工程	359
21.1 关注点分离	360
21.2 方面、连接点和切入点	363
21.3 采用方面的软件工程	365
21.3.1 面向关注点的需求工程	366
21.3.2 面向方面的设计和编程	368
21.3.3 检验和有效性验证	371
要点	372
进一步阅读材料	372
练习	373
参考书目	373

第四部分 软件管理

第 22 章 项目管理	376
22.1 风险管理	377
22.1.1 风险识别	378
22.1.2 风险分析	379

22.1.3 风险规划	380	24.3.2 程序审查	421
22.1.4 风险监控	381	24.4 软件度量和量度	422
22.2 人员管理	381	24.4.1 产品量度	425
22.3 团队协作	384	24.4.2 软件组件分析	426
22.3.1 成员挑选	386	24.4.3 度量歧义	427
22.3.2 小组的结构	387	要点	428
22.3.3 小组的沟通	388	进一步阅读材料	428
要点	389	练习	429
进一步阅读材料	390	参考书目	429
练习	390	第 25 章 配置管理	431
参考书目	390	25.1 变更管理	433
第 23 章 项目规划	392	25.2 版本管理	436
23.1 软件报价	393	25.3 系统构建	438
23.2 计划驱动的开发	394	25.4 发布版本管理	442
23.2.1 项目计划	395	要点	443
23.2.2 规划过程	395	进一步阅读材料	444
23.3 项目进度安排	397	练习	444
23.4 敏捷规划	400	参考书目	444
23.5 估算技术	402	第 26 章 过程改善	445
23.5.1 算法成本建模	402	26.1 过程改善过程	447
23.5.2 COCOMO II 模型	404	26.2 过程度量	448
23.5.3 项目的工期和人员配备	409	26.3 过程分析	450
要点	410	26.4 过程变更	452
进一步阅读材料	410	26.5 CMMI 过程改善框架	454
练习	410	26.5.1 分阶段的 CMMI 模型	456
参考书目	411	26.5.2 连续 CMMI 模型	458
第 24 章 质量管理	413	要点	458
24.1 软件质量	415	进一步阅读材料	459
24.2 软件标准	416	练习	459
24.3 复查与审查	419	参考书目	459
24.3.1 复查过程	420	术语表	461

软件工程导论

这一部分的目标是对软件工程做一般性的介绍。将介绍软件工程的一些重要概念，如软件过程和敏捷方法。描述软件开发中的基本活动，从初始的软件描述到系统进化。这部分的各章设计为支持软件工程一个学期的课程。

第1章是一个概述，介绍了专业软件工程并定义了一些软件工程的观念，也就软件工程职业道德方面的问题进行了简短的讨论。我认为软件工程师们认真思考所从事工作的深远影响是很重要的。这一章也介绍了贯穿本书使用的3个案例研究，分别是：保存正在接受心理健康问题治疗的病人记录的系统、治疗糖尿病患者的便携式胰岛素泵的控制系统和野外气象系统。

第2章和第3章分别介绍软件工程过程和敏捷开发。第2章介绍常用的软件过程模型，如瀑布模型，也介绍了这些软件过程中的基本活动。第3章增加了一个关于软件工程的敏捷开发方法的讨论。我喜欢使用极限编程作为敏捷方法的例子，这里也简要地介绍了 Scrum 的内容。

这部分剩余的各章是对在第2章中介绍的软件过程活动的深入阐述。第4章讲解需求工程的最为重要的话题，对系统应该做什么的需求给予了定义。第5章介绍使用 UML 语言对系统建模，专注于用例图、类图、时序图和状态图这些软件系统建模的方法。第6章介绍体系结构设计、体系结构的重要性以及在软件设计中体系结构模式的使用。

第7章介绍面向对象的设计和模式的使用，还介绍了重要的实现问题——复用、配置管理、宿主机-目标机的开发，并且讨论了开源开发。第8章主要介绍软件测试，从系统开发时的单元测试到软件的发布测试。此外，第8章还讨论了测试驱动开发的使用，这是在敏捷方法中率先使用的一种方法，现在已经得到广泛应用。最后，第9章概述软件进化问题，包括进化过程、软件维护和遗留系统管理。

概述

目标

本章的目标是介绍软件工程的概​​念，并为理解本书其他部分内容提供一个框架。读完本章，你将了解以下内容：

- 什么是软件工程，为什么它很重要；
- 开发不同类型的软件系统可能需要不同的软件工程方法；
- 道德和职业问题对于软件工程的重要性；
- 介绍 3 个不同类型的软件系统，这 3 个系统将作为贯穿全书的例子。

现代社会离不开软件。国家基础设施和公共建设都是由基于计算机的系统控制，大多数的电子产品都有计算机和控制软件。工业制造和分销已经完全计算机化了，金融系统也是这样。娱乐业，包括音乐产业、计算机游戏产业、电影和电视产业，也是一个软件密集型的产业。因此，软件工程对于一个国家和整个国际社会的运转都是必不可少的。

软件是抽象的、不可触摸的，它不受物质材料的限制，也不受物理定律或加工过程的制约，这一方面使软件工程得以简化，因为软件的潜能不受物理因素的限制；另一方面，由于缺乏自然约束，软件系统也就很容易变得极为复杂，理解它会很困难、改变它价格高昂。

从简单的嵌入式系统到复杂的全球信息系统，有很多不同类型的软件系统。正是由于不同的软件系统需要不同的技术，所以试图为软件工程寻求通用的符号系统、方法和技术是毫无意义的。开发一个机构信息系统和开发一个科学仪器的控制器是完全不同的。而这些系统都跟图形密集型的计算机游戏没有太多的共同点。所有这些应用都需要软件工程，但不是都需要相同的软件工程技术。

现在仍有许多有关软件项目出问题和“软件失败”的报道。软件工程因不能充分支持现代软件的开发而遭非议。然而，在我看来，这些所谓的软件失败源于以下两方面的原因：

1. 不断增长的需求 由于新的软件工程技术可以帮助我们构建更大更复杂的系统，用户的需要因而在发生改变。系统必须更快速地构建并交付；需要更大更复杂的系统；系统必须具备在以前看来不可能实现的功能。现有的软件工程方法已经不能应对新形势，而新的软件工程技术还有待于进一步发展。

2. 期望值太低 不采用软件工程的方法和技术去编写计算机程序相对来讲要容易一些。许多公司因为他们的产品和服务在逐步发展而在软件开发中随波逐流。他们通常不使用软件工程方法。结果导致他们的软件比预计的费用高且不可靠。因此我们需要更好的软件工程教育和实践来解决此类问题。

软件工程人员应该为自己所做出的成绩感到自豪。当然我们在开发复杂软件时还存在问题，但如果没有软件工程，我们就不能探索太空，也就没有因特网和现代的远程通信，各种形式的旅行就会很危险且花费很高。软件工程在它诞生以后的不长时间里就已做出了巨大贡献。我坚信：随着软件工程这门学科的不断成熟，它对 21 世纪的贡献将是不可估量的。



软件工程的历史

“软件工程”这一概念是在 1968 年召开的一个当时被称作“软件危机”的会议上首次提出的 (Naur 和 Randell, 1969)。当时, 单个的程序开发技术已经不能扩展从而应用到大型的、复杂的软件系统中。软件项目有时甚至要推迟几年才能完成, 而且比预计的费用高、不可靠、难以维护。

20 世纪 70 年代和 80 年代, 各种新的软件工程技术和方法都得到了发展, 例如结构化编程、信息隐藏和面向对象开发。工具和标准的符号系统得以研究和发展且现在得到了更广泛的使用。

[http: //www. SoftwareEngineering-9. com/Web/History/](http://www.SoftwareEngineering-9.com/Web/History/)

1.1 专业化软件开发

许多人都在编写程序。业务人员编写电子表格程序来简化工作, 科学家和工程师编写程序来处理实验数据, 业余爱好者为了自己的兴趣和爱好也编写程序。然而, 绝大多数的软件开发是个专业化的活动, 软件的开发是达到为了特定的业务目的, 是为了植入到其他的设备、作为软件产品, 例如信息系统、CAD 系统等。那些除了开发者外还有其他用户使用的专业化软件通常都是由团队开发而不是某个人独自完成的, 在其生命周期内要不断维护和修改。

软件工程的目的是支持专业化的软件开发, 而不是个体编程。它包括支持程序描述、设计和进化的相关技术, 而这些都不是个体软件开发所需要的。为了使大家对什么是软件工程有一个大体的认识, 图 1-1 总结了一些常见的问题。

问 题	答 案
什么是软件	计算机程序和相关文档。软件产品可针对特定客户开发或为通用市场开发
什么是优良软件的特点	好的软件应具有用户所需的功能与性能, 而且应该可维护、可靠、可用
什么是软件工程	软件工程是关于软件生产的各个方面的工程学科
什么是基本的软件工程活动	软件描述、软件开发、软件验证以及软件进化
软件工程和计算机科学有何区别	计算机科学侧重理论和基础, 而软件工程则侧重于软件开发和交付的实际活动
软件工程和系统工程有何区别	系统工程侧重基于计算机系统的开发的所有方面, 包括硬件、软件和过程工程。软件工程只是这个总体过程中的一部分
软件工程面临的主要挑战是什么	不断增长的多样性、减少交付时间以及开发可靠的软件的要求
什么是软件工程的成本	软件开发成本约占总成本的 60%, 测试成本占 40%。对于定制软件而言, 进化成本常常高于开发成本
什么才是最好的软件工程技术和方法	由于所有的软件项目都必须进行专业化的管理和开发, 所以不同的技术应适用于不同类型的系统。例如, 游戏开发需要一系列原型, 而安全要求极高的控制系统开发需要一个完整的和可分析的描述。因此, 你不能单纯地评判一种方法比另一种方法更好
Web 给软件工程带来了哪些不同	Web 带来了软件服务的可用性, 以及开发高品质的基于服务的分布式系统的可能性。基于 Web 的系统开发极大地促进了编程语言和软件复用的发展

图 1-1 关于软件的常见问题

许多人把软件等同于计算机程序, 其实这种理解是很狭隘的。在我们讨论软件工程时, 软件包括程序和所有使程序正确运行所需要的相关文档和配置信息。一个专业化开发的软件系统通

常远不止一个程序。系统通常包含一些单独的程序、用于设置这些程序的配置文件，可能还包括描述系统结构的系统文档和解释如何使用该系统的用户文档，以及告知用户下载最新产品信息的 Web 站点。

这是专业软件开发与业余软件开发的一个重要区别。如果你只是自己编写一个程序，且除你自己之外没有别的用户的话，你就不用写程序指南和设计文档等。然而，如果你的软件有别的用户，并且别的工程师会去修改它的话，你就必须提供除了程序源码之外的其他附带的信息。

软件工程人员关心的是软件产品（即能卖给客户的软件）的开发。软件产品有以下两类：

1. **通用软件产品** 由软件开发机构制作，在市场上公开销售，可以独立使用。这类软件产品有数据库软件、字处理软件、绘图软件以及工程管理工具等。还包括用于特定目的的所谓的“垂直”应用产品，如图书馆信息系统、财务系统等。

2. **定制软件产品** 这些产品受特定的客户委托，由软件承包商专门为这类客户开发。这类软件有电子设备的控制系统、特定的业务处理系统和空中交通管制系统等。

这两类产品的一个重要区别在于：在通用软件产品中，软件描述由开发者自己完成，而定制软件产品，其软件描述通常是由客户给出，开发者必须按客户要求开发。

然而，这两类产品之间的界线正在变得越来越模糊。现在更多的公司从一个通用软件产品开始进行定制处理，以满足特别客户的具体要求。企业资源规划（ERP）这类系统，如 SAP 系统，就是这种方法的一个最好见证。像这样的一个庞大而复杂的系统，需要通过嵌入一系列信息，比如说业务和操作规则以及各种报表等，以适应一个新企业。

软件除了提供相应的功能以外，作为一个产品它还有一系列相关的反映质量的属性。这些属性不直接涉及软件的功能，而是反映软件在执行时的行为以及源程序的结构、组织及相关的文档。软件对用户查询的响应时间和程序代码的可理解性就属于这类属性（有时称为非功能性属性）。

软件系统在具体应用中，用户可能会要求其具有特殊的属性。例如，银行系统必须安全可靠、交互式游戏必须响应快、电话交换系统必须可靠等，这些都是设计精良的软件系统必须具有的属性，这些属性归纳于图 1-2 中。这些属性也是专业化软件系统应具备的基本属性。

产品特性	描 述
可维护性	软件必须能够不断进化以满足客户的需求变化，这是软件产品最根本的特性，因为工作环境是不断变化的，软件也必然要跟着变化
可依赖性和安全性	软件可依赖性还包括一些特性：可靠性、保密性、安全性。可靠的软件在系统失败的情况下，也不会导致物理性损害和经济损失。有恶意的人员不能访问或破坏系统
有效性	软件不要浪费内存和处理器等系统资源，因而有效性应包括响应时间、处理时间和内存利用率等方面
可用性	软件必须简单易用，容易被用户接受。这就意味着，它必须是容易理解的、易用的并且和其他系统是兼容的

图 1-2 好软件的重要属性

1.1.1 软件工程

软件工程是一门工程学科，涉及软件生产的各个方面，从最初的系统描述一直到投入使用后的系统维护，都属于其学科范畴。在软件工程的定义中有两个关键词：

1. **工程学科** 干什么事情都离不开工程人员，他们既拥有一定的理论、方法和工具，又能

有选择地利用它们,即使在没有可用的理论和方法的情况下,也能够力求找出解决问题的方法。同时他们也认识到必须在机构或财政状况所允许的限度内工作,即在此限度内寻找解决办法。

2. 软件生产的各个方面 软件工程不仅涉及软件开发的技术过程,也涉及诸如软件项目管理以及对那些支持软件生产的工具、方法和理论的开发等活动。

工程都是要求在时间表和预算范围内获得所要求的品质的成果。这就要求我们在此有个折中,工程人员不能是完美主义者。人们为自己编写软件的时候往往是可以上面花费用任意的时间的。

总之,软件工程人员在其工作中运用的是系统的、有组织的方法,因为这种方法对于制作高质量的软件通常是最有效的。然而工程就是为各种情况选择最恰当的解决办法,因而更具创造力的、不太规范的开发方法在某些情况下可能是很有效的。不太规范的开发尤其适用于基于 Web 的系统开发,后者需要融合软件设计和图形设计技巧。

软件工程之所以重要有两方面的原因:

1. 个人和社会越来越多地依赖于先进的软件系统。这就需要我们能够既经济又快速地生产出可依赖和值得信赖的系统。

2. 从长远来看,运用软件工程方法和技术去开发软件系统比单纯为个人程序项目写程序更加便宜。对于大多数类型的系统来说,绝大多数的钱都花费在软件投入使用后对软件的变更上。

软件工程中系统化的方法有时候也叫软件过程。软件过程是指制作软件产品的一组活动及其结果。这些活动主要由软件工程人员完成。所有的软件过程都包含 4 项基本的活动,它们是:

1. 软件描述 客户和工程师定义所要生产的软件以及对其操作的一些约束。
2. 软件开发 软件得以设计和编程实现。
3. 软件有效性验证 软件经过检查以保证它就是客户所需要的。
4. 软件进化 软件随不同的客户和变化的市场需求而进行修改。

不同类型的系统需要不同的软件开发过程。举例来说,对于飞机上所使用的实时软件,我们就需要在开发开始之前做好全部的软件描述工作。而对于像电子商务这类系统,描述和编程往往是交织在一起的。因此,这些基本活动会以各种方式组织在一起,并且描述的详细程度也随软件类型的不同而不同。在第 2 章中我们还要详细讨论软件过程。

软件工程还涉及计算机科学和系统工程:

1. 计算机科学研究的是支撑计算机和软件系统的理论和方法,而软件工程则研究软件制作中的实际问题。正如电子工程师必须具有一定的物理学知识一样,软件工程人员同样必须具有一定的计算机科学知识。计算机科学理论通常更适用于相对较小的程序。对于大型的或是复杂的需要用软件解决的问题,计算机科学的经典理论不可能总是适用的。

2. 系统工程是研究有关复杂系统的开发和进化的方方面面,此类系统中软件起着重要的作用。因而,系统工程就涉及硬件部署、策略和过程设计、系统实施,也包括软件工程。系统工程人员的工作包括系统定义,定义它的总体体系结构,然后集成各个组件以完成整个系统。他们较少关注系统各组件(硬件、软件等)的工程问题。

如在下一节的讨论所述,我们将看到许多种不同类型的软件。没有一种通用的软件工程方法和技术适合于所有的软件。然而大体上却主要有 3 方面的问题影响着大多数类型的软件。

1. **异质性** 人们越来越要求系统像基于网络的分布式系统一样运行,而网络中包含不同类型的计算机和移动设备。除了在通用的计算机上运行之外,有些软件可能还需要在手机上运行。有时必须将新软件集成到遗留系统中,这些遗留系统可能是用其他语言写成的。这样带来的挑战是:必须开发新技术,制作可靠的软件,从而足以灵活应对这种多样性。

2. **业务和社会的变革** 随着新经济成长和新技术的不断涌现,业务和社会正在发生着前所

未有的快速变革。这对现存软件的变更和快速地开发出新软件提出新需求。很多传统的软件工程技术是费时的，新系统的交付往往远远滞后于预先的计划。因此技术必须进化，在不损及系统质量的前提下，缩短大型、复杂系统的交付时间。

3. 安全和可信 软件和我们生活的方方面面息息相关，最关键的一点是软件要让人们信得过。这对于那些通过网页或 Web 服务界面访问的远程软件系统来说尤其重要。我们必须确保有恶意的人员不能攻击软件，不会危及信息安全。

当然，以上问题也不是孤立的。例如，有时需要快速修改遗留系统，使之易于通过 Web 服务界面提供服务。为了应对这些挑战，我们需要有新的工具和技术，以及融合和使用现有软件工程方法的创新举措。

1.1.2 软件工程的多样性

软件工程是生产软件的系统化的方法，它考虑到了实际成本、进度、可靠性等问题，以及软件生产者和消费者的需要。怎样实施这个系统化的方法取决于软件开发机构、软件类型和开发过程中的人员。没有一个通用的软件工程方法和技术适合所有的系统和公司。多样的软件工程方法和工具已经进化了 50 多年。

也许决定使用哪种软件工程方法和技术主要取决于要开发的应用的类型。这里有许多不同类型的应用：

1. 独立的应用 这类应用运行在本地计算机上，比如 PC。它们拥有所有必要的功能但不需要连接到网络上。这类应用有 PC 上的办公软件、CAD、图片处理软件等。

2. 以交易为基础的交互式应用 这类应用在远程计算机上执行，用户通过自己的 PC 或终端进行访问。显然，这些应用包括 Web 应用，例如电子商务应用，你可以通过和一个远程系统交互购买商品或服务。这类应用还包括业务系统，一个企业通过 Web 浏览器或者特殊的客户端程序以及基于云服务允许用户访问它们的系统，比如邮件和照片共享。交互式系统通常包含大规模的数据存储，这些数据在每一次交易中被访问和刷新。

3. 嵌入式控制系统 这类应用有一个软件控制系统控制和管理硬件设备。嵌入式系统在数量上远远多过其他类型的系统。包括移动电话中使用的软件、汽车上控制防抱死的软件，以及微波炉上控制烹饪过程的软件。

4. 批处理系统 设计这类业务系统用来处理大批量的数据。它们处理大量的单个输入以创建相应的输出。这类系统包括定期账单系统，比如手机账单系统和工资支付系统。

5. 娱乐系统 这类系统主要是个人用户用于娱乐。大多数的这类系统是各种各样的游戏。这类系统所提供的交互质量是娱乐系统和其他系统的一个重要区别。

6. 建模和仿真系统 科学家和工程师用这类系统模拟物理过程或环境，这里有许多独立且相互交互的对象。通常是计算密集型的，需要高性能的并行系统才能运行。

7. 数据采集系统 这类系统用一些传感器从环境中采集数据并发送这些数据给其他系统进行处理。软件必须能同传感器进行交互且通常是安装在恶劣环境中，比如安装在发动机内部或者是荒郊野外。

8. 集成的系统 这类系统是由许多其他的软件系统所构成的。其中一些为通用软件产品，比如电子表格程序。其他的一些可能是专门为这个环境编写的软件。

当然，这些类型软件之间的边界是模糊的。如果你开发一个手机游戏，你必须与手机软件开发者一样考虑相同的约束（电源、硬件交互）。批处理系统通常和基于 Web 的系统配合使用。例如，一个公司的差旅费报销可能通过 Web 应用提交，但却在批处理应用中处理以便每月支付。

每种类型的软件都有不同的特征，因此需要使用不同的软件工程技术。例如，汽车上的嵌入

式控制系统是对安全性要求极高的，在车上安装时要烧制到 ROM 中。因此一旦发生改变就会十分昂贵。这样的系统应得到全面的核查和校验，以确保售后因为软件问题被召回的可能性最小。用户的交互在这里是很少的（或许根本就没有），因此没有必要使用依赖于用户接口原型的开发过程。

基于 Web 的系统，可能适合用迭代式开发和交付，使得系统可以包含很多可复用的组件。然而，这个技术可能就不适合用在集成式系统上，集成式系统要求预先详细定义系统之间的交互，以便每个系统都可以单独进行开发。

不管怎样，还是有很多软件工程的基本方法适用于所有类型的软件系统：

1. 应使用有管理的和理解了的开发过程进行开发。软件开发机构应规划它们的开发过程，并清楚地知道应产出什么以及什么时候完工。当然，对于不同类型的软件使用不同的开发过程。

2. 可依赖性和性能对所有类型的系统来说都很重要。软件应该如所期待的那样表现，没有失败且在用户需要的时候是可用的。它应该是操作安全的，只要可能，它应该是信息安全的，能抵御来自外部的攻击。系统应是高效的且不会浪费资源。

3. 理解和管理系统描述和需求（系统应该做的是什）是很重要的。你必须知道不同的客户和用户的期望是什么，然后你必须管理这些期望以便在预算范围内按期交付一个有用的系统。

4. 你应该尽可能高效地使用当前存在的资源。这就意味着，你应该在适当的地方复用已开发的软件，而不是重新写一个新软件。

这些开发过程的基本概念、可依赖性、需求、管理和复用是这本书的重要主题。不同的方法以不同的方式反映这些概念，但它们是所有专业化软件开发的基础。

你应该注意到这些概念没有涉及实现和编程。在本书中，不涉及具体的编程技术是因为不同的系统使用的技术是有很大的不同的。例如，基于 Web 的系统编程使用如 Ruby 一类的脚本语言，而这类语言完全不适合嵌入式系统工程。

1.1.3 软件工程和 Web

万维网的发展已经对我们的生活产生了深远的影响。最初，Web 只是一种可以广泛被访问的信息源，对软件系统几乎没有影响。这些系统在本地上计算机上运行，且只是由同一机构内部的人访问。2000 年左右，Web 开始演化，浏览器添加了越来越多的功能。这就意味着基于 Web 的系统的开发已经不再是建立一种特殊的用户界面，而是允许用户通过浏览器访问这些系统。这就导致了大量新系统产品的开发，通过万维网访问的方式提供新服务。这些系统通常是依赖广告赞助，广告显示在用户屏幕上，不需要用户直接付费。

除了这些系统产品之外，开发可以运行小程序和运行一些本地处理程序的 Web 浏览器，导致了业务和机构软件的演化。不同于将写好的软件部署在 PC 上运行，这些软件现在部署在 Web 服务器上。因为不必在每一台 PC 上安装软件，这就使得变更和升级软件已经非常便宜。由于用户界面的开发格外昂贵，这就更节约了成本。因此，只要可能的话，许多公司就会将软件系统放到 Web 服务器上，通过 Web 浏览器完成各种先前的业务。

基于 Web 系统开发的下一个阶段是 Web 服务的概念。Web 服务是软件的一个组成部分，它提供特殊有用的服务并通过 Web 访问。应用程序就是通过集成可能由不同公司提供的 Web 服务所构造的。在原理上，应用程序在运行时的连接是动态的，这样一个应用就可以在每次执行的时候使用不同的 Web 服务。第 19 章将介绍这种软件开发技术。

最近几年产生了一种观点——“软件就是一种服务”。人们提出的方案是，软件不再在本地上计算机上运行，而是将它放在所谓的“计算云”里，通过因特网访问。假如你用了诸如网络邮件的服务，你就用了基于云的系统。一个计算云是被很多用户共享的一个庞大的互联在一起的

计算机集群系统。用户只需依照使用软件的次数付费而不需要购买软件，或是可以免费使用，但会在屏幕上看到广告。

因此，Web 的出现使业务软件在组成上发生了显著的改变。在没有 Web 之前，绝大多数的业务应用都是大一统的，单个的程序运行在单个计算机上或是一个计算机群上。通信是在本地和机构内部进行的。现在，软件是高度分布式的，有时要横贯整个地球。业务应用不再是从头开始编写程序，而是涉及对组件和程序的大规模复用。

显然，这种软件组织的显著变化导致了以网络为基础的系统设计的变化。比如：

1. 软件复用已经成为构建基于 Web 的系统的主要技术。当你在构造这样的系统时就需要考虑怎样从已有的软件组件和系统开始工作。

2. 现在人们普遍认识到，提前指定这些系统的所有需求是不切实际的。应逐步开发和交付这种基于 Web 的系统。

3. 用户界面受到 Web 浏览器能力的约束。尽管类似于 AJAX (Holdener, 2008) 这样的技术可以在浏览器内创建丰富的界面，但这些技术目前还是非常难用。用 Web 表格加本地脚本的方式更常用一些。在基于 Web 的系统上的应用界面通常比专门为 PC 系列产品专门设计的用户界面要差。

前面一节讨论的软件工程的基本思想，同样也适用于基于 Web 的软件，如同其他类型的软件一样。在 20 世纪从大型软件开发所获得的经验仍然可以指导我们构造基于 Web 的软件系统。

1.2 软件工程师的职业道德

和其他工程人员一样，软件工程师必须承认他们的工作不仅仅是技术的应用，还要担负许多责任。他们的工作是在法律和社会认可的框架内完成的。软件工程师要想受人尊敬，其行为就必须合乎道德，必须有责任感。

软件工程师必须坚持诚实正直的行为准则，这是不言而喻的。他们不能用掌握的知识和技能做不诚实的事情，更不能给软件工程行业抹黑。然而，在有些方面，某些行为没有法律加以规范，只能靠职业道德来约束，这种约束是软弱无力的。包括：

1. **保密** 工程人员必须严格保守雇主或客户的机密，而不管是否签署了保密协议。

2. **工作能力** 工程人员应该实事求是地表述自己的工作能力，不应有意接受超出自己能力的工作。

3. **知识产权** 工程人员应当知晓有关专利权、著作权等知识产权的地方法律，必须谨慎行事，确保雇主和客户的知识产权受到保护。

4. **计算机滥用** 软件工程师不应运用自己的技能滥用他人的计算机。滥用计算机有时对他人影响不大（如在雇主的机器上玩游戏），但有些时候后果非常严重（传播病毒）。

在这一方面职业协会和机构肩负重任。ACM、IEEE（电气和电子工程师协会）和英国计算机协会等组织颁布了职业行为准则或职业道德准则，凡是加入这些组织的成员必须严格遵守。这些行为准则只涉及基本的道德行为。

ACM 和 IEEE 还联合推出了一个关于职业道德和职业行为的准则，有两个版本：一个比较简短，见图 1-3；另一个较长（Gottterbarn 等，1999），增加了一些细节和要义，并把简写版中放在后面的理论阐述进行归纳后放在了头两段：

计算机在商业、工业、政府、医药、教育、娱乐和整个社会中的核心作用日渐突出。软件工程师是直接参与或讲授软件系统的分析、描述、设计、开发、认证、维护和测试的人员。基于他们在软件系统开发中的地位，软件工程师可能将事情做好也可能做坏，还可能让他人或影响他人将事情做好或做坏。为了最大限度地保证他们的工作是有益的，软件工程师必须保证

使软件工程业成为对社会有益的、受人尊敬的行业。基于以上保证，软件工程人员应当遵守下面的道德和职业行为准则。

本行为准则包括八项基本原则，针对包括软件工程行业的从业者、教育者、管理者、监督者、政策制定者、接受培训者和学生在内的职业软件工程人员。这八条原则阐明了个人、团队和机构之间职业道德上的责任关系以及他们在其中应该履行的基本义务。每一原则的条款都表述了这些关系中的一些义务。这些义务既基于软件工程人员的人性，也对那些受他们的工作和软件工程实践的独特环境影响的人们表示出特别的关怀。本准则把这些内容规定在任何一个自称或渴望成为软件工程人员的义务中。

软件工程职业道德和职业行为准则

(ACM/IEEE-CS 联合制定以规范软件工程行业的职业道德和职业行为)

序 言

准则的简写版把对软件工程人员的要求做了高度抽象的概括，完整版中的条款把这些要求细化，并给出了实例，用以规范软件工程专业人员的工作方式。没有这些总体要求，所有的细节都是教条而又枯燥的；而没有这些细节，总体要求就会变成空洞的高调。只有把二者紧密结合才能形成有机的行为准则。

软件工程人员应当作出承诺，使软件的分析、描述、设计、开发、测试和维护等工作对社会有益且受人尊重。基于对公众健康、安全和福利的考虑，软件工程人员应当遵守以下 8 条原则：

1. 公众感——软件工程人员应始终与公众利益保持一致。
2. 客户和雇主——软件工程人员应当在与公众利益保持一致的前提下，保证客户和雇主的最大利益。
3. 产品——软件工程人员应当保证他们的产品及其相关附件达到尽可能高的行业标准。
4. 判断力——软件工程人员应当具备公正和独立的职业判断力。
5. 管理——软件工程管理者和领导者应当维护并倡导合乎道德的有关软件开发和维护的管理方法。
6. 职业感——软件工程人员应当弘扬职业正义感和荣誉感，尊重社会公众利益。
7. 同事——软件工程人员应当公平地对待和协助每一位同事。
8. 自己——软件工程人员应当毕生学习专业知识，倡导合乎职业道德的职业活动方式。

图 1-3 ACM/IEEE 职业道德准则 (© IEEE/ACM 1999)

因为不同的人有不同的观点和目标，产生道义困境是难免的。打个比方，假如你原则上不赞成公司高级管理层的决策，你该怎么办？显然，这要看每个人的个性和不赞成的原因了。是在团队内部坚持自己的观点并据理力争，还是坚持原则毅然辞职，哪种做法是最好的呢？如果你觉得一个软件项目有问题，你会选择什么时机向管理层报告呢？如果只是在怀疑，这时向管理层报告未免有点过敏；如果拖很长时间才报告管理层，则有可能延误了解决难题的时机。

在我们的职业生涯中，每个人都会面临这些困惑，幸运的是，在多数情况下，这些困惑要么不严重，要么不难解决。如果这些困惑解不开，工程人员也许要面对另一个问题——辞职，但这样做会影响到其他人，如影响同伴或自己的孩子。

当雇主的行为不合乎道德时，职业工程人员的处境尤为艰难。比如，一个公司负责开发对安全性要求极高的系统，由于时间紧张而篡改了安全的有效性验证记录，这时工程人员的责任是保守机密、提醒客户注意，还是以一定的方式向客户披露交付的系统可能不安全？

这里的问题在于安全不是绝对的。尽管系统没有按照事先给定的标准去验证安全有效性，但这些标准很可能过于苛刻。系统或许在整个生命周期过程中一直都能安全运行。还可能有一种情况：即使有效性得到了准确验证，最后系统仍然可能失败而导致灾难发生。早期披露这些问题将使雇主和其他雇员蒙受损失，如果隐瞒这些问题又会对其他人极为不利。

在这个问题上必须有自己的主见。上述情况中潜在的灾难、灾难的严重程度以及灾难的受

害者，这些因素都将影响决定的做出。如果情况非常危险，就应该通过一定的方式披露出来，但这时还应该同时尊重雇主的权利。

另一个道德问题是军事项目和核项目的参与。许多人已强烈地意识到这个问题，不想参与到军事项目中。有的人则愿意开发军事系统，但不能涉及武器系统。然而，另有一些人认为国防具有压倒一切的重要性，反对参与武器系统的开发是毫无道理的。

在上述情况中，雇主和所有雇员事先相互沟通各自的观点非常重要。若以一个机构的形式参与军事系统或核系统的开发，应该确保每个雇员都自愿接受工作安排。同样，如果雇员已经明确表示他们不愿意参加这类项目，雇主就不应该在日后给他们施加压力。

随着软件密集型系统越来越多地渗透到人们的日常工作和生活中，普通领域中道德和职业上的责任问题越来越受到关注。这个问题的讨论可以从哲学的角度研究道德问题的基本原理，而软件工程职业道德的研究可以参照这些基本原理。Laudon (1995) 就采用这种方法，而 Huff 和 Martin (1995) 则基本不用这种方法。Johnson 在关于计算机道义方面的论述 (2001) 也涉及此话题，他是从哲学角度论起的。

我认为用哲学的研究方法太过抽象，很难与我们的日常生活联系起来。我比较推崇更具体的研究方法，并将其与行为和实践准则相结合。道德问题的研究最好要联系软件工程的实际，而不是将其作为一个孤立的问题来研究。因此，本书中没有抽象的道德问题的讨论，只在适当的地方讨论基本的道德问题，如练习中的一些例子。

1.3 案例研究

在本书中，为了介绍清楚软件工程的观念，我用了来自 3 种不同类型的系统的实例，并将其贯穿全书。我之所以没有使用单一案例是因为，这本书要传递的一个关键信息就是软件工程的实践取决于要构造的系统的类型。因此，在解释概念的时候我针对不同的概念选择了合适的例子，这些概念包括如安全性和可依赖性、系统建模、复用等。

用于案例的 3 种类型的系统分别是：

1. 嵌入式系统——系统的软件控制硬件设备并嵌入在这个设备中。嵌入式系统的典型问题包括物理尺寸、响应性、电源管理等。本书嵌入式系统的例子是一个软件系统控制的医疗设备。

2. 信息系统——这个系统的主要目的是管理和提供对信息数据库的访问服务。信息系统的主要问题包括信息安全性、可用性、隐私和维护数据的完整性。信息系统的例子是一个医疗数据系统。

3. 基于传感器的数据采集系统——这个系统的主要目的是从多个传感器收集数据并以适当的方式处理数据。这类系统的关键需求是可依赖性；甚至是在极端环境条件下的可靠运行、可维护性。数据采集系统的例子是一个野外气象站。

本章对这些系统做了大致的介绍，更多的信息可从网上得到。

1.3.1 胰岛素泵控制系统

胰岛素泵是一个仿真胰腺（一种体内组织）运转的医疗系统。此系统的软件控制部分是一个嵌入式系统，它从传感器收集数据，然后控制泵输送指定剂量的胰岛素给患者。

糖尿病患者使用这个系统。糖尿病是一种常见病症，是由于人体无法产生足够数量的胰岛素而引起的。胰岛素在血液中起到促进葡萄糖新陈代谢的作用。糖尿病的传统治疗方法是长期规律地注射人工胰岛素。通过使用一种外部仪器测量糖尿病患者的血糖值，从而计算所需要注射的胰岛素剂量。

这个治疗方法存在的问题是，血液中的胰岛素浓度不仅与血液中的葡萄糖浓度相关，还与最后一次注射胰岛素的时间有关。这样有可能导致血糖浓度太低（当胰岛素太多时），或血糖浓度太高（当胰岛素太少时）。短时间内的低血糖是一种比较严重的情况，会导致暂时的脑功能障碍，最后失去知觉甚至死亡。长期处于高血糖则会导致眼睛损伤、肾损伤和心脏问题。

目前在开发微型传感器方面取得的进步使得自动胰岛素传送系统开发成为可能。系统监控血糖浓度，根据需要输送适当的胰岛素。这样的胰岛素输送系统已经在临床中得到使用了。在将来有可能将这样的系统永久地植入糖尿病患者体内。

该系统使用一个植入在人体内的微传感器来测量一些血液参数，这些参数与血糖浓度成正比。这些参数被送到泵控制器。控制器计算血糖浓度，得出胰岛素需要量，然后向一个小型化的泵发送信号使之通过持久连接的针头输送胰岛素。

图 1-4 给出了胰岛素泵的硬件组件和组成结构。要理解本书中系统的实际例子，关键是要知道血液传感器测量血液在不同情况下的电传导率，而这些值是和血糖浓度相关的。胰岛素泵输送一个单位的胰岛素就会使控制器发送一个简单的脉冲。因此输送 10 个单位的胰岛素，控制器就会向泵发送 10 个脉冲。图 1-5 是一个 UML 活动模型，描述了怎样将输入的血糖浓度值转换为驱动胰岛素泵的命令序列。

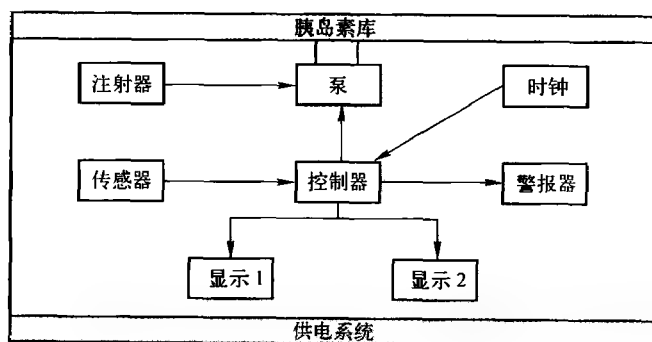


图 1-4 胰岛素泵的结构

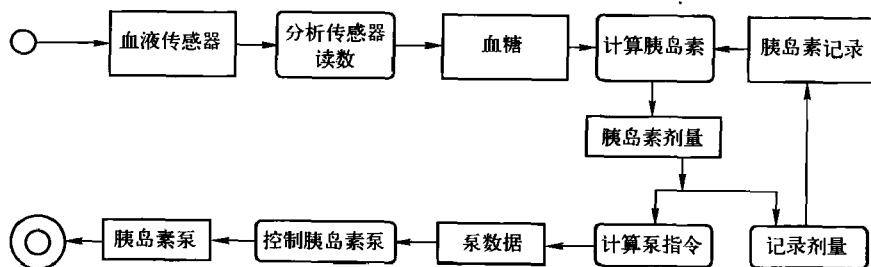


图 1-5 胰岛素泵的活动模型

显然，这是个安全性要求极高的系统。若此系统的操作失败，将会危及病人的健康，病人甚至因血糖浓度过低或者过高引起昏迷。因而在这个胰岛素输送系统中，有两个高级别的紧要需求是系统所必须达到的：

1. 当需要输入胰岛素的时候这个系统能够输送胰岛素。
2. 系统必须能可靠地运行，并根据当前血糖浓度输入正确剂量的胰岛素。

因此系统的设计和实现必须确保能满足这些需求。本书将在后面的章节介绍更多的具体需求并讨论如何确保系统的安全性。

1.3.2 用于心理健康治疗的患者信息系统

支持心理健康治疗的患者信息系统是一个医疗信息系统，它维护着心理疾病患者接受治疗的信息。大多数有心理问题的人不需要住院治疗，但是需要定期去那些详细了解他们病情的专科诊所看医生。为了方便病人看病，这些诊所不只是在医院里，也有可能开在当地私人诊所或社区中心。

MHC-PMS（心理健康治疗－患者管理系统）是打算在诊所使用的一个信息系统。它利用一个患者信息的中心数据库，也可以运行在 PC 计算机上，这样它就可以在即使没有安全的网络连接的地方运行了。在有安全的网络连接时，他们可以从数据库中获取用户信息并复制到本地，当没有网络连接时，系统可以利用下载的患者信息。这个系统不是一个完备的医疗数据系统，因而不需要维护其他方面的信息。然而，它可以与其他诊所信息系统互通并交换数据。图 1-6 说明了 MHC-PMS 的组成。

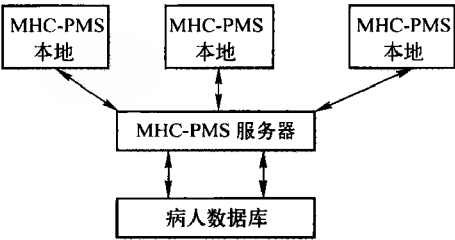


图 1-6 MHC-PMS 的组成

MHC-PMS 有两个总体目标：

- 1. 生成管理信息，使医疗卫生管理部门能够据此评估本地和政府在此方面的执行情况。
- 2. 为医疗服务人员提供及时的相关信息以保证对病人的治疗。

精神卫生问题的特性决定了患者常常错过预约时间，故意或意外遗失处方和药品，不听医生嘱咐，并对医务人员提出无理要求，不期而至等。在少数情况中，他们可能对自己和他人造成危害。他们可能经常更换住址，或长期或短期地离家出走。如果病人是危险的，他们可能需要被“隔离”，即限定在一个安全的医院接受治疗和观察。

系统的用户包括医务人员如医生、护士和家庭护士（上门检查健康状况的护士）。非医务人员包括负责预约的接待员、负责维护系统医疗数据的工作人员，以及负责生成报告的行政人员。

系统负责记录病人信息（姓名、地址、年龄、亲属等）、会诊（日期、责任医生、对病人的主观印象等）、病人状况和治疗方案。系统定期产生报告供医务工作人员和卫生部门管理者使用。通常医务工作人员关注的是单个病人的信息，而用于管理目的的报告往往是隐去患者姓名等信息的，关注统计意义上的患者状况、治疗的花费等。

这个系统的主要特征是：

- 1. **病例管理** 医生可以为病人创建记录、在系统中编辑信息、查看病人历史等。系统支持数据汇总，这样某个先前未接触过病人的医生也可以快速了解此病人的主要问题和当前的治疗情况。
- 2. **病人监测** 系统定期监测那些正在接受治疗的病人的记录，若探测到可疑问题就会发出提醒。因此，若某个病人好长时间都没有看医生了，系统就会发出通知。此监测系统最重要的一个特点是能够对强制隔离的病人保持跟踪，以确保在正确的时间能够对其进行合乎法律要求的例行检查。

3. **管理报告** 系统产生月度管理报告, 显示每个诊所接治的病人数目、进入和离开护理系统的病人数目、受到强制隔离的病人数目、处方药物的使用情况及其价格等。

有两项法律条款影响该系统。一个是有关个人信息保密性的数据保护法, 另一个是有关强制监禁被认为对病人自己和其他人会造成伤害的病人的心理健康法。心理健康在这方面是很独特的, 因为它是唯一的可以违背病人的意愿将其拘留的医学专业。这是受到非常严格的立法保护的。MHC-PMS 系统的目的之一就是确保工作人员总是能够按照法律的规定工作, 且他们对病人的所有处理得以记录下来并能够在必要的时候经得起司法审查。

像所有的医疗系统一样, 隐私是一个要求极高的系统需求。病人信息是保密的, 不能暴露给除相关医护人员和病人自己之外的任何人。MHC-PMS 也是一个安全性要求极高的一个系统。一些精神疾病可导致病人自杀或者对其他人造成人身伤害。系统应尽可能向医护人员警示潜在有自杀倾向或者有危害倾向的病人。

系统的总体设计必须同时考虑隐私和安全两个方面的需求。系统必须是有效的, 不然的话其安全性就会大打折扣, 使得医生不能及时为病人拿出治疗方案。这里有一个潜在的冲突, 即当系统数据只有单个拷贝时, 隐私是最容易维护的。然而, 为确保在服务器失败或没有连接网络时系统的有效性, 会存在和维护多份数据拷贝。本书后面章节会讨论这些需求间的平衡问题。

1.3.3 野外气象站

为了监控偏远地区的气候变化、提高气象预报的准确度, 那些幅员辽阔的国家的政府会在偏远地区部署几百个气象站。这些气象站通过一组装置来采集气象数据, 比如温度、气压、光照、降雨、风速和风向。

野外气象站只是一个更大系统的一部分 (见图 1-7), 该大系统是一个从气象站采集数据并能将其提供给其他系统处理的气象信息系统。图 1-7 中的系统包括:

1. **气象站系统** 该子系统负责收集气象数据, 做一些初始处理, 然后传输给数据管理系统。
2. **数据管理与存档系统** 该子系统从野外气象站收集数据, 执行数据处理与分析, 将数据存储为容易被其他子系统如天气预报系统检索的格式。
3. **气象站维护系统** 该子系统可以通过卫星与所有野外气象站通信, 监控它们的运行状态, 并报告出现的问题。还可以更新这些气象站上的嵌入式软件。当某个野外气象站系统出问题时, 此系统还可以远程控制出问题的野外气象站系统。

在图 1-7 中, 使用 UML 包的符号来表示每个系统都是一个组件的集合, 并且采用 UML 标记《system》描述所有的独立系统。包之间的关联表示包与包之间存在信息交换, 但目前我们没必要对它们做更详细的描述。

每一个气象站都有许多采集天气数据的仪器, 比如风速、风向、气温、气压、24 小时降雨量等。所有这些设备都是在软件系统的控制下周期性地读入并管理所采集到的数据。

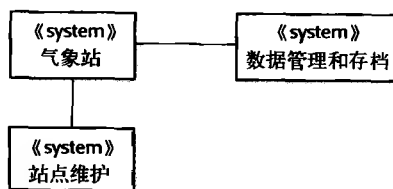


图 1-7 气象站的环境

气象站系统进行气象数据观察的频率是很高的, 比如对温度的测量需每分钟测量一次。然而, 由于卫星带宽相对较窄, 气象站需要在本地对数据进行一些处理和存储。然后当数据收集系统请求数据时, 提交存储在本地的数据。假如因为某个因素导致通信连接不成功, 气象站在本地继续保留数据, 直到通信恢复。

每个气象站都是独立电池供电的, 而且完全是自我管理的。没有外部供电或有线电缆存在。所有的通信都是通过相对慢速的卫星连接。气象站必须包含自我充电的机制, 比如太阳能或风能。由于它们是部署在野外的, 直接暴露在各种恶劣环境条件下, 还有可能被动物毁坏。因此气

象站软件不能仅仅只是做到数据采集。它还必须做到以下几点：

1. 监控仪器、电源、通信硬件，并向管理系统报告故障。
 2. 管理系统电源，确保电池在环境条件允许的情况下得到充电，也确保在恶劣天气情况下，比如大风天气，及时关闭发电机以免受到破坏。
 3. 允许动态配置，在部分软件版本更新时，或者是当系统发生失败而切换备份装置时。
- 因为气象站必须是自我管理和无人看管的，这就意味着尽管数据采集功能本身相对简单，但软件的安装是复杂的。

要点

- 软件工程是一门涉及软件生产的各个方面的一门工程学科。
- 软件产品不仅是程序，还包括相关文档。软件产品的基本属性是可维护性、可靠性、信息安全性、效率以及可接受性。
- 软件过程包括开发软件产品过程中的所有活动。软件过程中的活动主要有：软件描述、开发、有效验证和进化。
- 软件工程的基本概念普遍适用于所有类型的系统开发。这些基本概念包括软件过程、可靠性、信息安全性、需求以及复用。
- 世界上存在着很多类型的系统。每一种类型的系统的开发都需要一种与之相适应的软件工程工具和技术。几乎不存在普适的神奇的软件设计和实现技术。
- 软件工程的基本思想适用于所有的软件系统。这些基本思想包括有管理的软件过程、软件的可依赖性和信息安全性、需求工程和软件复用。
- 软件工程人员对软件工程行业和整个社会负有责任，不应该只关心技术问题。
- 职业协会颁布的行为准则规定了一系列协会成员应该遵守的行为标准。

进一步阅读材料

《No silver bullet: Essence and accidents of software engineering》，尽管这篇论文发表的时间较早，但仍然是一般性介绍软件工程问题的好论文。在过去的 13 年中，这篇论文的基本要点没有改变（F. P. Brooks, IEEE Computer, 20 (4), April 1987）。<http://doi.ieeecomputersociety.org/10.1109/MC.1987.1663532>。

《Software Engineering Code of Ethics is approved》，这篇文章介绍了制定 ACM/IEEE 职业道德准则的背景，这个准则包括一个简写版和一个较长版本（D. Gotterbarn, K. Miller, and S. Rogerson, Comm. ACM, October 1999）。<http://portal.acm.org/citation.cfm?doid=317665.317682>。

《Professional Issues in Software Engineering》，这是一本讨论法律、职业和道德问题的好书。较之其他理论性太强的课本，作者更推崇其实用方法（F. Bott, A. Coleman, J. Eaton and D. Rowland, 3rd edition, 2000, Taylor & Francis）。

《IEEE Software, March/April 2002》，这是一本专门介绍基于 Web 的软件开发的杂志。这一领域发展得非常快，因而有一些文章有一点过时，但大部分还是相关的（IEEE Software, 19 (2), 2002）。<http://www2.computer.org/portal/web/software>。

《A View of 20th and 21st Century Software Engineering》这本书是第一代最杰出的一些软件工程师对软件工程的回顾和展望。Barry Boehm 定义了永恒的软件工程原则，但也提出建议认为一些常用的做法已经过时（B. Boehm, Proc. 28th Software Engineering Conf., Shanghai, 2006）。<http://doi.ieeecomputersociety.org/10.1145/1134285.1134288>。

《Software Engineering Ethics》是 IEEE Computer 的专刊，里面有很多的论文（IEEE Computer,

42 (6), June 2009)。

练习

- 1.1 解释为什么专业化软件不仅仅包括为用户所开发的程序。
- 1.2 通用软件产品开发和定制软件开发之间有什么不同？这在实际应用中对通用软件产品用户意味着什么？
- 1.3 软件产品应该具有的4个重要属性是哪些？另外列举出4个可能有意义的属性。
- 1.4 除了异质性挑战、业务和社会的变革、安全和可信，说出软件工程在21世纪的可能面临的其他问题和挑战。（提示：比如环境。）
- 1.5 参照1.1.2节讨论的应用类型，举例解释为什么设计和开发不同类型的应用需要专门的软件工程技术。
- 1.6 解释为什么软件工程的基本思想适用于所有的软件系统。
- 1.7 解释Web的普遍使用是怎么改变软件系统的。
- 1.8 职业工程人员是否应该和医生或律师一样要颁发资格证书？讨论一下。
- 1.9 对图1-3的ACM/IEEE职业道德准则中的每一条款，举出一个恰当的例子加以说明。
- 1.10 为了反恐，很多国家正计划开发或正在开发一种对其大量公民及其行动跟踪的计算机系统。显然这是侵犯个人隐私权的做法。对开发此类系统作伦理道德方面的讨论。

参考书目

- Gotterbarn, D., Miller, K. and Rogerson, S. (1999). Software Engineering Code of Ethics is Approved. *Comm. ACM*, **42** (10), 102–7.
- Holdener, A. T. (2008). *Ajax: The Definitive Guide*. Sebastopol, Ca.: O'Reilly and Associates.
- Huff, C. and Martin, C. D. (1995). Computing Consequences: A Framework for Teaching Ethical Computing. *Comm. ACM*, **38** (12), 75–84.
- Johnson, D. G. (2001). *Computer Ethics*. Englewood Cliffs, NJ: Prentice Hall.
- Laudon, K. (1995). Ethical Concepts and Information Technology. *Comm. ACM*, **38** (12), 33–9.
- Naur, P. and Randell, B. (1969). Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany. 7th to 11th October 1968.

软件过程

目标

本章的目标是介绍软件过程的思想——软件生产的一组互相连贯的活动。读完本章，你将了解以下内容：

- 软件过程和软件过程模型的概念；
- 了解3个一般的软件过程模型及何时使用它们；
- 了解软件需求工程、软件开发、测试和进化中所涉及的基本过程活动；
- 理解为什么软件过程要有效地组织以应对软件需求和设计上的变更；
- 了解 Rational 统一过程（Rational Unified Process）是如何集成好的软件过程实践来产生一个可适应的软件过程。

软件过程是一组引发软件产品生产的活动。这些活动可能是使用像 Java 和 C 这样的标准编程语言从头开始的一步一步的软件开发。然而，业务应用没有必要采用这种做法。现在业务软件通常是通过扩展和修改现有的系统、或通过配置和集成商业现货软件或系统组件而获得。

虽然有许多不同的软件过程，但所有软件过程都必须具有4种对软件工程来说是基本的活动。它们是：

1. **软件描述** 必须定义软件的功能以及软件操作上的约束。
2. **软件设计和实现** 必须生产符合描述的软件。
3. **软件有效性验证** 软件必须得到有效性验证，即确保软件是客户所想要的。
4. **软件进化** 软件必须进化以满足不断变化的客户需要。

在很多时候，这些活动只是所有软件过程的一部分。当然，在实践中，它们本身就是复杂的，还包括子活动如需求验证、体系结构设计、单元测试等。还会有一些支持活动，如文档编写和软件配置管理。

当我们描述和讨论所谓的软件过程时，我们总是谈论过程中的活动（如数据模型的定义、设计用户界面等）以及这些活动的顺序。然而，除了这些活动以外，过程描述还会包括：

1. 产品，这是软件过程活动的结果。比如，体系结构设计活动的结果可能是得到一个软件体系结构的模型。
2. 角色，反映了人在软件过程中的职责。如项目经理、配置经理、程序员等。
3. 前置和后置条件，是指在一个过程活动执行的前后或产品生产的前后，陈述语句为真。如在体系结构设计开始之前，一个前置条件可能是客户已经认可了所有的需求；在此活动结束之后，后置条件可能是对体系结构描述的 UML 模型通过了审查。

软件过程是复杂的，且像所有智力和创造性过程一样，依赖于人们的决策和判断。并不存在什么理想的软件过程，大多数机构有自己的软件开发过程。软件过程在不断演化以充分利用机构中人的能力和所开发的系统的特殊性质。对于某些系统，如安全性要求极高的系统，需要一个结构化非常好的开发过程。对于业务系统，由于它需要适应迅速变化的需求，就需要一个不那么正式但很灵活的过程。

有时，软件过程被分为计划驱动过程和敏捷过程两类。计划驱动的过程是提前计划好所有的过

程活动，然后按计划去考核过程的执行。在敏捷过程中，如第3章介绍的，计划是增量式的，而且很容易根据不断变化的客户需求变更过程。正如Boehm和Turner（2003）所讨论过的那样，不同的技术适合于不同类型的软件。总之，你需要在计划驱动的过程和敏捷过程之间找到一个平衡。

虽然没有“理想”的软件过程，在许多机构中仍存在很大的改进软件过程的空间。有的过程可能包括落后的技术，或没有利用工业化的软件工程的很多最佳实践。甚至，许多机构在它们的软件开发过程中仍然没有采用软件工程方法。

软件过程改善可以通过过程标准化实现，这样将减少在一个机构中多样的软件过程的出现。这会带来诸多好处，如沟通的改善、缩短培训时间、使自动化的过程支持更经济。标准化也是在引入新的软件工程方法和技术以及好的软件工程实践方面非常重要的第一步。第26章将深入探讨软件过程改善这一主题。

2.1 软件过程模型

正如第1章所讨论的，软件过程模型是软件过程的简化表示。每个过程模型都是从一个特定的侧面表现软件过程，所以只提供过程的部分信息。例如过程活动模型表现了这些活动和它们的顺序，但是可能表现不出人们在这些活动中的角色。这一节中将介绍几个非常一般的过程模型（有时也叫做“过程范型”），并从体系结构的角度给出这些过程模型，即只关心过程的框架而不注重其特别活动的细节。

这些一般模型不是软件过程的唯一性描述，而是对软件过程一种有用的抽象，能用来解释软件开发的不同方法。你可以把它们看做过程框架，通过扩展和调整以创建更多专门的软件工程过程。

本章讨论的过程模型是：

1. **瀑布模型** 该模型将基本的过程活动、描述、开发、有效性验证和进化，看成是一些界限分明的独立的过程阶段，例如，需求描述阶段、软件设计阶段、实现阶段、测试阶段，等等。

2. **增量式开发** 该方法使得描述活动、开发活动和有效性验证活动交织在一起。系统的开发是建立一系列的版本（增量），每个版本添加部分功能到先前的版本中。

3. **面向复用的软件工程** 该方法是基于已存在的大量可复用的组件。系统开发过程着重于集成这些组件到新系统中，而非从头开发。

这3个模型相互不排斥，而且经常一起使用，尤其是对大型系统的开发。对于大型系统，综合瀑布和增量开发模型的优点是有意义的。你必须知道系统的核心需求，设计系统的软件体系结构以支持这些需求。这是不能增量式开发的。在更大系统中的子系统可以使用不同的开发方法。对于那些理解的很好的系统部分可以用基于瀑布模型的过程来描述和开发，而那些很难提前描述清楚的部分，如用户界面，就总是要用增量式开发方法。

2.1.1 瀑布模型

最初发表的软件开发过程模型起源于更一般的系统工程过程（Royce，1970），该模型如图2-1所示。因为瀑布从一个阶段到另一个阶段，这个模型因此以“瀑布模型”闻名，也可以看成是软件的生命周期模型。瀑布模型是计划驱动的软件过程的一个例子——理论上，在开始工作之前，你必须对所有的过程活动制订计划并给出进度安排。

瀑布模型中的主要阶段直接映射基本的开发活动：

1. **需求分析和定义** 通过咨询系统用户建立系统的服务、约束和目标。并对其详细定义形成系统描述。

2. **系统和软件设计** 系统设计过程通过建立系统的总体体系结构将需求区分为硬件需求和软件需求。软件设计包括识别和描述一些基本的软件系统抽象及其之间的关系。

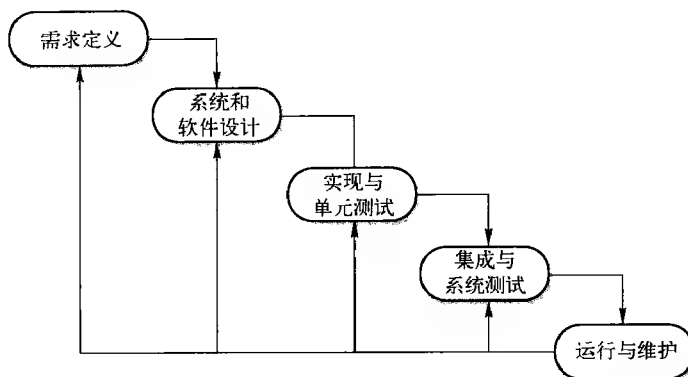


图 2-1 瀑布模型

3. **实现和单元测试** 在此阶段，将软件设计实现为一组程序或程序单元。单元测试就是检验每个单元是否符合其描述。

4. **集成和系统测试** 集成单个的程序单元或一组程序，并对系统整体进行测试以确保其满足了软件的需求。在测试之后，软件系统将交付给客户使用。

5. **运行和维护** 正常情况下（虽然不是必须的），这是一个具有最长生命周期的阶段。系统被安装并且投入实际的使用中。维护包括改正那些在早期各阶段末被发现的错误，改善系统各个单元的实现，并当新的需求出现时提高系统的服务能力。

原则上，每个阶段的结果是一个或多个经过核准的文档。直到上一个阶段完成，下一阶段才能启动。在实际过程中，这些阶段经常是重叠和彼此间有信息交换的。在设计阶段，可能发现需求中的问题；在编程阶段，又会发现设计当中的问题，以此类推。软件过程不是一个简单的线性模型，它包括对开发活动的多个反复。每一阶段产生的文档在后续的阶段都可能被修改以反映发生的变化。

因为生成和确认文档的成本很高，因而反复是昂贵的而且十分费时。因此，在经过少量的反复之后，要冻结部分开发过程，例如描述部分，继续进行后面的开发阶段。剩下的问题留着以后解决，或者忽略掉、或者在编程中想办法绕过去。这种对需求的冻结会使需求相当不成熟，这又意味着系统不能满足用户的需要。当设计上的问题通过一些编程的小技巧来解决时，系统的良好结构又会遭到破坏。

在最后的生命周期阶段（运行和维护阶段），软件进入使用状态。最初的软件需求中的错误和省略部分这时暴露无遗。设计阶段和编程阶段的错误此时也都浮现出来，同时对一些新的功能的需要也表现出来了。因此，系统必须进化以保持实用。实现这些变更（软件维护）可能需要对一些或所有的早先过程阶段进行重复。



“净室”软件工程

形式化开发过程的一个实例是净室（Cleanroom）过程，最初由 IBM 公司提出的。在净室过程中每一个软件增量都要给出形式化描述，然后此描述经过变换得以实现。软件的正确性通过形式化方法得以证明。在此开发过程中不进行单元缺陷测试，而系统测试的重心集中在评估系统的可靠性上。

该净化过程的目标是生产零缺陷软件，使得所交付的系统具有高度的可靠性。

<http://www.SoftwareEngineering-9.com/Web/Cleanroom/>

瀑布模型是与其他工程过程模型相一致的，在它的每个阶段都要生成文档。这使得过程是可见的，项目经理能够根据项目计划监控项目的过程。它的主要问题在于它将项目生硬地分解成这些清晰的阶段。关于需求的责任和义务一定要在过程的早期阶段清晰界定，而这又意味它对用户需求变更的响应较困难。

所以只有在对需求了解得好，而且在系统开发过程中不太可能发生重大改变的时候，适合采用瀑布模型。毕竟，瀑布模型反映了在其他工程项目中使用的一类过程的模型。由于在整个项目中它很容易结合通用的管理模式进行管理，基于该方法的软件过程仍然广泛应用于软件开发。

瀑布模型的一个重要变形是形式化系统开发。针对系统描述创建其数学模型，然后采用能保持一致性的数学变换对该数学模型进行加工，直到产生可执行代码。基于对所构造的数学变换是正确的假设，你就可以有力地证明用这种方式生成的代码是与其描述相一致的。

形式化的开发过程，如基于B方法（Schneider, 2001；Wordsworth, 1996）特别适用于有严格安全性、可靠性或信息安全性需求的系统的开发。形式化方法简化了安全用例和信息安全用例的生产。它向客户或行业管理者证明该系统实际上满足了其安全性和信息安全性的需求。

基于形式变换的过程通常只用于开发安全性或信息安全性要求极高的一类系统。这需要非常专业的知识和技能。对于大多数系统，在系统开发中应用这一过程不会比其他方法带来明显的成本优势。

2.1.2 增量式开发

增量式开发的思想是先开发出一个初始的实现，给用户使用并听取用户的使用意见和建议，通过对多个版本的不断修改直到产生一个充分的系统（见图2-2）。描述、开发和有效性验证等活动不是分离的而是交织在一起。同时让这些活动之间都能得到快速的反馈信息传递。

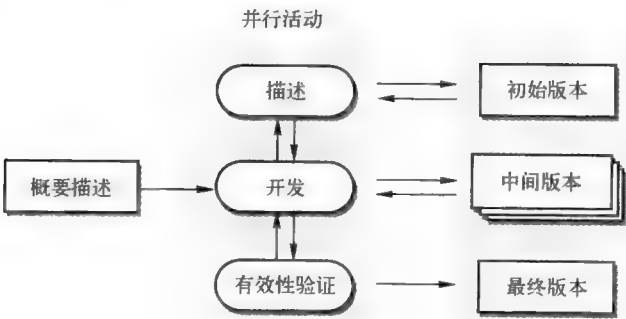


图 2-2 增量式开发

增量式软件开发，是敏捷方法的一个基本部分，对于商务、电子商务和个人系统来说更加适合。增量式开发反映了我们解决问题的方法。我们很少提前制定出完整的问题解决方案，而是逐步地逼近解决方案，当我们意识到错误的时候进行回溯。通过增量式地开发软件，在开发过程中可以更经济、更容易对软件变更做出响应。

系统的每一个增量或版本包括用户需要的一部分功能。通常，系统的早期增量包括最重要或最紧急的功能需求。这就意味着在早期开发阶段，用户可以在相对早地评估系统，看它是否满足需要。假如不满足需要，只有目前的增量需要改变，或许，有新的功能被发现并为下个增量做准备。

增量式开发较之瀑布模型有3个重要优点：

- 1. 降低了适应用户需求变更的成本。重新分析和修改文档的工作量较之瀑布模型要少很多。

2. 在开发过程中更容易得到用户对于已做的开发工作的反馈意见。用户可以评价软件的现实版本，并可以看到已经实现了多少。这比让用户从软件设计文档中判断工程进度要好很多。
3. 使更快地交付和部署有用的软件到客户方变成了可能，虽然不是所有的功能都已经包含在内。相比于瀑布模型，用户可以更早地使用软件并创造商业价值。



增量式开发的问题

尽管增量式开发有很多优点，但它也不是没有问题。它的困难的主要原因是，大型机构的官僚办事程序随着时间的推移已经根深蒂固，有可能这些办事程序与更加非形式化的迭代和敏捷过程不匹配。

有时，这些办事程序是有其存在的充分理由的，如，可能有些程序是为了确保在软件开发过程中能正确执行外部法规（如美国的萨班斯会计法规）。不可能改变这些程序，因此过程冲突是不可避免的。

<http://www.SoftwareEngineering-9.com/Web/IncrementalDev/>

增量式开发在形态上可能有多种，它现在是最常用的应用系统开发方法。这种方法即可以是计划驱动的也可以是敏捷方法的，或更常用的做法是对这些方法的混合运用。在计划驱动的方法中，系统增量是提前定义好了的；如果采用敏捷方法，最初的增量是定义好了的，但是后面的增量的开发是完全取决于项目的进展情况以及客户的优先选择。

从管理的角度来看，增量式方法存在两个问题：

1. 过程不可见。管理者需要通过经常性的可交付文档来把握进度，如果系统开发速度太快，要产生反映系统每个版本的文档就很不划算。
2. 伴随着新的增量的添加，系统结构在逐渐退化。除非投入时间和金钱用在重构系统结构上以改善软件，否则定期的变更会损坏系统的结构。随着时间的推移，越往后变更系统越困难，而且成本也将逐渐上升。

对于那些大型的、生命周期很长的系统，不同的团队负责开发系统的不同部分，进化式开发的问题就变得非常突出了。大型系统需要一个稳定的框架或体系结构，而负责开发不同部分的团队需要根据体系结构明确指定其职责。这需要提前制订计划，而不是增量地开发。

可以进化式地开发一个系统，并展示给客户，征求客户意见，而不是实际地交付给客户并在客户的环境下部署运行它。增量式的交付和部署意味着软件在一个真实的运转过程中使用。通常这是不太可能的，因为试验一个新软件会扰乱正常的业务过程。2.3.2节将讨论增量式交付的优点和缺点。

2.1.3 面向复用的软件工程

在大多数的软件项目中，都存在一定程度的软件复用。当人们注意到某项目中的设计或代码是与当前项目中所需要的部分相像的时候，复用就自然地发生了。人们搜寻这些东西，而后根据需要进行修改它们，再将其纳入自己的系统中来。

这样随意性的复用并没有考虑所采用的开发过程。然而，在21世纪，注重复用现存软件的开发过程得到了广泛采用。面向复用的方法依赖于存在大量可复用的软件组件以及能组合这些组件的集成框架。有时，这些组件本身就是一个系统（COTS即商业现货系统），能提供专门的功能，例如字处理或制表软件。

基于复用开发的一般过程模型如图2-3所示。尽管初始需求描述阶段和有效性验证阶段与其

他过程差不多，面向复用过程的中间阶段是不一样的。这些阶段是：

1. **组件分析** 给出需求描述，然后搜寻能满足需求的组件。通常情况是，没有正好合适的组件以供选择，能得到的组件往往只提供所需要的部分功能。

2. **需求修改** 在这个阶段，根据得到的组件信息分析需求，然后修改需求以反映可得到的组件。当需求修改无法做到的时候，就需要重新进入组件分析活动以搜索其他可能的替代方案。

3. **使用复用的系统设计** 在这个阶段，设计系统的框架或者重复使用一个已存在的框架。设计者分析那些将被重复使用的组件，并组织框架使之适应这些组件。当某些可复用的组件不能得到时，必须重新设计一些新的软件。

4. **开发和集成** 当组件不能买到时就需要自己开发，然后集成这些自己开发的组件和现货组件，使之成为一个整体。在这个模型中，系统集成与其说是一个独立的活动，不如说已经成为开发过程的一个部分。

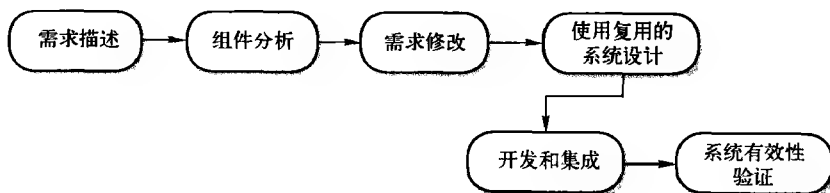


图 2-3 面向复用的软件工程

3 种类型的软件组件可能用于面向复用的过程：

1. 通过标准服务开发的 Web 服务，可用于远程调用。
2. 对象的集合，作为一个包和组件框架，如 .NET 或者 J2EE 等集成在一起。
3. 独立的软件系统，通过配置在特定的环境下使用。

面向复用的模型的明显优势是它减少了需要开发的软件数量，从而降低了软件开发成本，同时也降低了开发中的风险。通常也可使软件快速地交付。然而，需求妥协是不可避免的，而且这可能又导致一个不符合用户真正需要的系统。此外，对系统进化的控制也将失效，因为可复用的组件新版本可能是不受机构控制的。

软件复用是非常重要的，本书的第三部分专门讲解软件复用。第 16 章将介绍软件复用的一般问题和 COTS 复用，第 17 章和第 18 章介绍基于组件的软件工程，第 19 章将介绍面向服务的系统。

2.2 过程活动

真正的软件过程是交织着技术、协作、管理等内容的一个活动序列，围绕着一个总的目标，即实现系统的描述、设计、实现和测试。软件开发人员在工作中使用各种不同类型的软件工具。工具对不同类型的文档编辑以及管理大型项目中所产生的庞大数量的详细信息是特别有用的。

对于不同的开发过程，描述、开发、验证和进化这 4 个基本过程活动的组织是不同的。在瀑布模型中，它们是顺序组织的，而在增量式开发中它们是交织在一起的。这些活动如何执行，是依赖于软件类型、人员以及机构的组织结构的。例如，在极限编程中，描述是写在卡片上的。测试是可执行的且在程序本身存在之前就已经完成了。进化可能包括对系统的重组和重构。

2.2.1 软件描述

软件描述或需求工程是理解和定义系统需要提供哪些服务，以及找出开发和运行中受到哪

些约束。需求工程是软件过程中一个特别关键的阶段，这个阶段的错误将不可避免地带来系统设计和实现阶段的后续问题。

需求工程过程（见图 2-4）目标是生产一个达成一致意见的需求文档，定义能满足信息持有者需求的系统。通常需求在这个文档中被分成两个层次表述。最终用户和客户需要高层的需求描述；而系统开发人员需要比较详细的系统描述。



软件开发工具

软件开发工具（有时也叫计算机辅助软件工程或 CASE 工具）是用来支持软件工程过程活动的程序，这些工具包括如：设计编辑器、数据字典、编译器、调试器、系统生成工具等。

软件工具通过自动化某些过程活动和提供正被开发的软件的信息来支持软件过程。可以自动化的活动的例子包括：

- 作为需求描述和软件设计中的一部分的图形系统模型的开发。
- 从这些图形模型生成代码。
- 通过由用户交互创建的图形界面描述来产生用户界面。
- 通过提供执行的程序的信息进行程序调试。
- 程序的自动转换，将使用旧语言版本的程序转换成一个新版本语言的程序。

多种工具可能被整合在一个被称为交互式软件环境的框架中，也成为 IDE。这就为不同的工具提供了一个共同的可用的设施集合，使之更容易在集成的方式下通信和运行。人们广泛使用 ICLIPSE IDE，它嵌入了很多不同类型的软件工具。

<http://www.SoftwareEngineering-9.com/Web/CASE/>

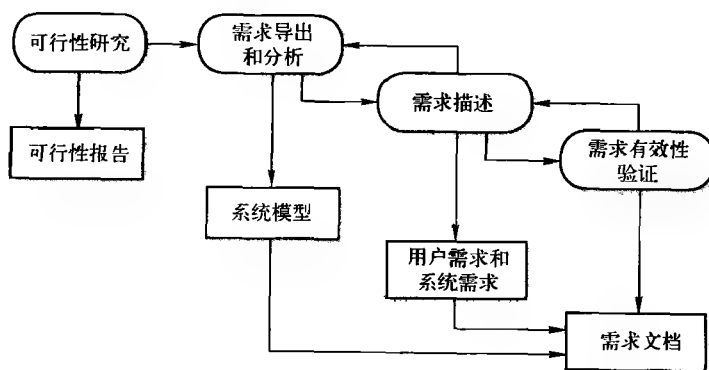


图 2-4 需求工程过程

需求工程过程有 4 个主要的阶段：

1. **可行性研究** 指明现有的软件、硬件技术能否实现用户对新系统的要求。从业务角度来决定系统开发是否划算以及在预算范围内能否开发出来。可行性研究应该是相对来讲花钱少且快速完成的。结果应该得出结论，该系统是否值得进行更细致的分析。

2. **需求导出和分析** 这是一个通过对现有系统分析、与潜在用户和购买者讨论、进行任务分析等导出系统需求的过程。也可能需要开发一个或多个不同的系统模型和原型。这些都会帮助分析员了解所要描述的系统。

3. **需求描述** 就是把在分析活动中收集的信息以文档的形式确定下来。在这个文档中有两类需求。用户需求是从客户和最终用户角度对系统需求的抽象描述；系统需求是对系统要提供

的功能的详尽描述。

4. 需求有效性验证 该活动检查需求的现实性、一致性和完备性。在这个过程中，肯定会发现需求文档中的错误，必须加以改正以解决问题。

当然，需求过程中的各项活动并不是严格按顺序进行的。在定义和描述期间，需求分析继续进行，这样在整个需求工程过程中不断有新的需求出现。因此，分析、定义和描述是交替进行的。在敏捷方法中，如极限编程，需求是按照用户的优先选择增量式开发的，需求的导出来自于作为开发团队一份子的用户。

2.2.2 软件设计和实现

软件开发的实现阶段是把系统描述转换成一个可运行的系统的过程。它总是包含设计和编程，但是，如果用增量式开发方法，可能还包含对软件描述的精炼过程。

软件设计是对实现软件的结构、系统的数据、系统组件间的接口以及所用的算法的描述。设计者不可能一次就能完成一个完整的设计，这是一个多次反复的过程。在设计过程中要不断添加设计要素和设计细节，并对先前的设计方案进行修正。

图 2-5 是这个过程的抽象模型，显示了设计过程的输入、过程活动和作为这部分输出的文档。从图 2-5 中可以看出设计过程中的各个阶段是顺序的。事实上，设计过程中的活动是交替进行的。从一个阶段到另一个阶段的反馈及其引发的返工在所有的设计过程中都是不可避免的。

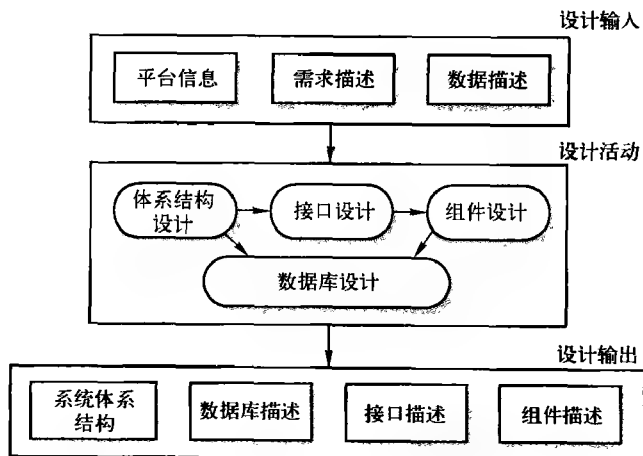


图 2-5 设计过程的通用模型

绝大多数软件有面向其他软件系统的接口。包括操作系统、数据库、中间件和其他应用系统。这些软件系统构成了“软件平台”，也即软件执行的环境。关于此平台的信息是设计过程的基本输入，因为设计者必须考虑怎样才能使它与软件环境集成得最好。需求描述是软件必须实现的功能、性能和可靠性需求的说明。如果是一个处理现有数据的系统，那么数据描述就可能包括在平台描述里；否则，数据描述就必须是设计过程的输入，以定义系统数据结构。

设计过程中的活动是会有所不同的，这取决于要开发系统的类型。例如，实时系统需要进行时序设计，但可能不包括数据库，因此就没有数据库设计。图 2-5 显示了信息系统设计过程中 4 个活动：

1. 体系结构设计 识别系统总体结构、基本组件（有时候也叫子系统或模块）、它们之间的关系以及它们是怎样分布的。

2. **接口设计** 定义系统组件间的接口。接口的描述必须是无二义的。在有精确接口定义的前提下, 组件不必知道其他组件的具体实现即可使用它们。一旦接口描述达成一致意见, 组件就可以并行设计和开发了。

3. **组件设计** 针对每个系统组件设计它的运行方式。这可能是对预期功能的一个简单声明, 留给程序员去做具体的设计。也可能是对复用组件或是某个细化的设计模型所做的一系列的变更。设计模型可用于自动生成一个实现。

4. **数据库设计** 设计系统数据结构, 以及如何在数据库中表示这些数据结构。此处的工作又一次取决于是否复用现有数据库或创建一个新的数据库。



结构化方法

结构化方法是软件设计的一种方法, 定义待开发的图形模型作为设计过程的一部分。这个方法可能也会定义适用于每一个模型开发的过程以及规则。结构化方法为系统产生标准化的文档, 尤其是为缺乏经验的和非专业的开发者提供了一个开发框架。

<http://www.SoftwareEngineering-9.com/Web/Structured-methods/>

这些活动产生一组设计输出, 这也在图 2-5 中给出。这些输出的细节和表示有很大不同。对于要求极高的系统, 必须要有详细的设计文档给出简洁和准确的描述。假如用模型驱动方法, 这些输出可能大多是图形。而用敏捷开发方法, 设计过程的输出可能不会有单独的描述文档, 而是在程序代码中直接表示。

设计中的结构化方法出现在 20 世纪 70 年代和 80 年代, 是 UML 和面向对象设计 (Budgen, 2003) 的先导。它们依赖于创建系统的图形模型, 在很多情况下, 从这些模型能自动生成代码。模型驱动的开发 (MDD) 或模型驱动的工程 (Schmidt, 2006), 即在不同的抽象水平上创建软件的模型, 这是结构化方法的一个进化。在 MDD 中, 更强调体系结构模型——将实现独立的模型和具体实现的模型相分离。模型设计得充分详细, 因此可以由此生成可执行系统。第 5 章将介绍该方法。

在系统设计过程之后就是实现系统的程序开发过程。尽管有些程序, 如安全性能要求极高的系统的程序, 在系统任何实现进行之前就要完成所有的详细设计, 而更一般情况是, 后续的设计阶段和程序开发是交织在一起的。通过软件开发工具可以从一个设计直接得到一个程序的框架, 其中包括定义和实现界面的代码, 且在多数情况下, 开发者只需要增加每个程序组件的工作细节即可。

程序设计因人而异, 通常没有统一的模式。一些程序设计者从已经理解得很好的组件开始做起, 然后再做不太熟悉的组件。其他人可能会采取相反的方法, 把熟悉的组件留到最后, 因为他们已经知道该如何开发这些组件。一些开发者喜欢先定义过程中要用到的数据, 然后使用这些数据来驱动程序的开发; 而另外一些人则是对数据现用现定义。

通常, 程序设计者要对自己开发的程序进行测试。这时程序中的一些明显的错误就会暴露出来并得到根除, 该过程叫做调试。错误检测和程序调试不是一回事, 检测是要发现存在的错误, 而调试是要定位并改正这些错误。

程序员进行程序调试首先要对程序的行为有一个最初的预计, 然后执行程序看输出结果是否同预期的一样。调试过程需要程序员一步步地手动跟踪代码, 同时会需要一些测试用例来定位问题。一些交互式调试工具, 能够显示程序中变量的中间结果, 也能跟踪程序的执行语句, 可以用来支持程序的调试过程。

2.2.3 软件有效性验证

软件有效性验证，通常也称为检验和有效性验证（V&V），是要看系统是否符合它的描述以及系统是否符合客户的预期。程序测试，即用模拟测试数据运行系统，是最基本的有效性验证技术。有效性验证技术也包括在从用户需求定义到程序开发的每个软件过程阶段进行检查过程（比如，查阅和复审）。由于测试的主导地位，绝大多数的有效性验证成本发生在系统完成过程中和完成之后。

除了很小的程序，系统都不是作为一个孤立的单元来测试的。图2-6给出了一个分成3个阶段的测试过程，先是系统组件测试，其次是系统集成测试，最后是用客户数据对系统的测试。理想情况是，在系统集成前，组件缺陷和接口问题能尽早地发现。然而，一旦缺陷被发现，程序就需要进行调试，而这又会引起一系列测试阶段的反复。有的组件错误只有当集成时才暴露出来，因此，这个过程是一个反复的过程，不断会有信息从后继的阶段反馈到前面的阶段。



图2-6 测试各阶段

测试过程中的阶段包括：

1. **组件（或单元）测试** 由开发系统的人员对组成系统的组件进行测试。每个组件都单独测试，而不受其他系统组件的影响。组件可能是简单的实体，如一个函数或对象类、或者是这些实体的一个相关集合。通常用自动化测试工具，如JUnit（Massol 和 Husted，2003），它能在新版本的组件创建的时候对其重新测试。

2. **系统测试** 集成组件形成完整的系统。这个过程主要是关注无法预测的组件间交互和组件界面问题所引发的错误。也关注系统是否满足了功能上和非功能上的需求，并测试系统的总体特性。对于大型系统来说，这可能是一个多阶段的过程，需要对组件所构成的子系统进行测试，然后测试由这些子系统所构成的最终系统。

3. **接收测试** 这是系统在接收并运行之前进行的最后阶段测试。这个阶段不再是用模拟数据来测试系统，而是用客户提供的真实数据测试系统。真实数据能以不同的方式测试系统，所以能暴露出系统需求定义中的错误和遗漏。接收测试还能发现系统需求中的类问题，即系统的设施不能满足用户的需要或者系统性能是无法接受的。

通常，组件开发和测试是交叉进行的。程序员自己构造用来测试的数据，在代码开发过程中增量地进行测试。这是一个经济的测试方法，程序设计者最了解组件，因此是产生测试数据的最佳人选。

如果采用的是增量式开发方法，每一个增量在开发的时候就要得到测试，这种测试根据对增量的需求来设计。对于极限编程，测试和需求都先于开发过程。这帮助测试者和开发者理解需求并保证不会由于设计测试案例而拖延时间。

当采用计划驱动的软件过程（如开发安全要求性极高的系统），测试是测试计划集驱动的。一个独立的测试团队需要有一个拟定的测试计划并按照这个测试计划来工作，制订测试计划的依据是系统描述和设计。图2-7说明测试计划是如何把测试和开发活动联系起来的。有时候这也叫V开发模型。

接收测试有时称为“alpha测试”。定制系统是为特定客户开发的。测试过程持续进行，指

导开发者和客户双方都承认交付的系统是对需求的可接受的实现。

当一个系统要作为软件产品在市场上销售时，所要进行的测试称为“beta 测试”。“beta 测试”就是将系统交付给所有愿意使用该系统的潜在客户。他们有义务报告系统中的问题，以使产品面向实际使用，“beta 测试”能暴露系统开发者无法预见的错误。通过这个反馈，系统被修改并且发布另外一个测试版本或正式销售版本。

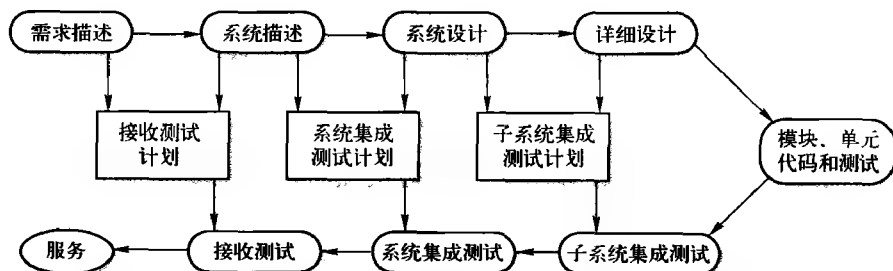


图 2-7 计划驱动软件过程中的测试阶段

2.2.4 软件进化

软件系统的柔性（灵活性）是在大型复杂系统中具有越来越多的软件的主要理由之一。一旦硬件生产的决定已经做出，再对硬件设计做出变更将是非常不经济的。然而，作为软件，变更可以发生在系统开发之中或之后的任何时间里。这些变更也可能非常昂贵，但对比系统硬件的变更仍然便宜得多。

自有软件开发以来，就有软件开发过程和软件进化（软件维护）过程之分。软件开发是一个有创造性的活动，从一个初始的概念发展成一个可实际工作的系统的整个过程一直充满了智慧和灵感。相反，人们通常认为软件维护是单调无趣的。虽然“维护”的成本经常是最初开发成本的数倍，但是维护过程通常不被认为有什么挑战性。

现在看来，这种划分越来越不恰当了。现在完全从头开发的系统很少，将软件系统的开发和维护看成是一个连续体显得更有意义。因此，不再将软件工程看成是开发和维护两个完全独立的过程，而是将其看成是一个进化过程，即软件在其生命周期内不断地随着需求的变更而变更的进化式过程。这个进化式过程如图 2-8 所示。

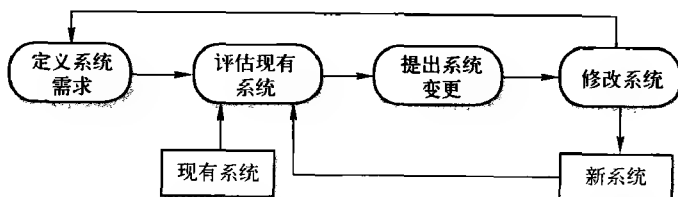


图 2-8 系统进化

2.3 应对变更

对于所有大型项目来说，变更是无法避免的。系统需求是在变化的，因为采购系统的工作要屈从于外部压力和管理权力的更替。当有新技术可用的时候，新的设计和实现方法就会出现。因此不管用什么样的软件过程模型，能在软件开发过程中及时处理变更是很必要的。

变更增加了软件开发的成本，因为这通常意味着已完成的工作要重做。这叫做返工。例如，

假如已经分析了系统需求之间的关系，之后又定义了新的需求，那么部分或者全部需求分析就必须重做。也可能要重新设计系统，交付新的需求，变更所有已开发的程序，并重新设计系统。

有两个相关的方法会用于降低返工成本：

1. **变更避免** 软件过程包括一些能够在重大返工发生之前预测变更的活动。例如，原型系统的开发，要先给客户看系统的一些重要特征。客户可以试用原型，并在花费高额的软件生产成本之前重新定义需求。

2. **变更容忍** 所设计的过程使得变更以相对较低的成本得到处理。这通常需要增量开发。提出的变更可能是在还没有开发的增量上实现。假如不是这样的，只需要更改单个增量（系统的一小部分）来适应变更。

这一节将介绍两种应对变更系统需求的方法。它们是：

1. **系统原型** 快速开发一个系统版本或者是系统的一个部分，以检验客户需求和某些设计决定的可行性。它支持变更避免，因为它允许客户试用在正式交付之前的系统，并重新审视和定义需求。因而在交付正式系统之后，客户提出的需求变更的数目将会降低。

2. **增量交付** 系统增量地交付给用户，给用户评审和试用。它支持变更避免和变更容忍。避免了交付不成熟的需求实现，并允许在之后的增量中进行改变且将成本控制在较低水平。

重构的概念，即改进程序的结构和组织，也是变更容忍的一个重要方法。将在第3章介绍重构，放在敏捷方法内容之中。

2.3.1 原型构造

原型是一个软件系统的最初版本，用于验证概念、试用设计选项、发现更多的问题和可能的解决方法。原型的快速迭代开发是必要的，这样就可以控制成本，系统信息持有者也可以在软件过程的早期试用系统原型。

软件原型可以用在软件开发过程中，帮助预计可能需要的变更：

1. 在需求工程过程中，原型有助于启发和验证系统需求。
2. 在系统设计过程中，原型可用于探索特定软件的解决方案，并支持用户接口设计。

系统原型允许用户思考系统怎样才能很好地帮助他们工作。他们可能对需求有新的想法、发现软件的长处和短处。然后他们可能提出新的需求。此外，当开发原型时，可能会发现已提出需求的错误和遗漏。需求描述中的功能看起来是可用的、定义是对的。但是，当结合了其他的功能，用户通常会发现他们最初的想法是不对的、不完整的。然后，受来自他们的需求理解的变化，可能需要修改系统描述。

系统设计时用系统原型，旨在验证所提出的设计的可行性。例如，一个数据库原型设计和测试，可检查它是否高效地支持了最常见的用户查询。原型也是用户接口设计过程必不可少的一部分。因为用户界面的动态性，文字描述和图都足以表达用户的界面需求。所以，让最终用户介入的快速原型构造是开发软件系统的图形用户界面的唯一明智方法。

原型开发的过程模型如图2-9所示。从开发过程的开始就应明确地建立开发原型的目标。这些可能是用户界面的原型、验证系统功能需求，或对管理者证明应用的可行性。同一个原型不能满足所有的目标。假如有遗漏的目标没有言明，则管理者或最终用户可能会误解原型的功能。因而原型的开发可能没有发挥预期的效果。

过程的下一阶段就是决定什么要放入原型系统中，或许更重要的是，什么不要放入原型系统中。为了减少原型开发的成本并缩短交付时间，可能要在原型中放弃一些功能。可能会放松类似于响应时间、内存利用率等非功能的需求。错误处理和错误管理可能被忽略，除非原型的目标是构建用户界面。程序的可依赖性和质量方面的标准可能会降低。

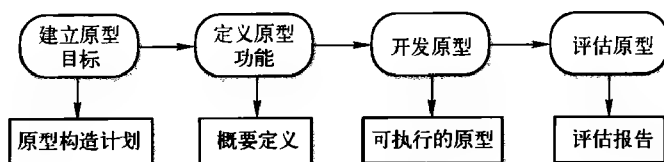


图 2-9 原型开发的过程

最后一个阶段是原型评估。这个阶段必须安排用户培训，应根据原型的目标制订一个评估计划。用户需要一段时间来适应新系统并习惯于正常的使用模式。一旦他们自然地使用了系统，他们将发现错误的和被遗漏的需求。

原型开发的一般问题是，原型不一定必须与最终系统有一样的使用方式。原型的测试者可能不是系统的典型用户。原型评估期间的训练时间可能不够。如果原型是缓慢的，评估人员可以调整工作方式，避免这些反应时间过慢的系统特征。当最终系统有更好的响应时，用户可以用不同的方式使用它。

有时，管理者可能会强迫开发人员把一个抛弃式原型当做正式系统交付给用户，特别是当软件的最终版本交付发生延期时。这种做法通常是很不明智的：

1. 不可能调整原型以满足非功能性的要求，如性能、安全性、鲁棒性和可靠性需求。这些都会原型开发时被忽略。
2. 开发过程中的快速更改必然意味着原型是没有文档的。唯一的设计描述就是原型的代码。这不利于长期的维护。
3. 原型开发过程中的变更可能会破坏系统的结构。系统的维护将会很困难，而且很昂贵。
4. 在原型开发中机构的质量标准通常是被放松了的。

不强求原型是可执行的。基于纸质模型系统的用户界面（Rettig, 1994）可以有效地帮助用户细化界面设计，并通过使用脚本（也称“使用场景”）工作。它的开发很便宜，而且可以在短时间内构造完成。对该技术的扩展就是 Oz 原型精灵，这里只开发用户界面。用户和这个界面交互，但他们的请求被传递给一个人，该人员解释此请求并输出相应的响应。

2.3.2 增量式交付

增量式开发（如图 2-10 所示）是软件开发的一种方法，所开发的某些增量逐步交付给用户，并在用户的操作环境下运行。在增量式开发过程中，客户大概地提出系统需要提供的服务，指明哪些服务是最重要的，哪些是最不重要的。此时，一系列交付增量被确定，每个增量提供系统功能的一个子集。对增量中服务的分配取决于服务优先次序。具有最高优先权的服务首先被交付。

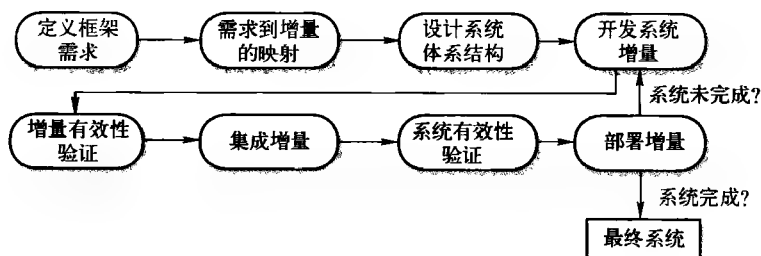


图 2-10 增量式交付

一旦确定了系统增量，在最先交付的增量中将详细地定义服务的需求，而这个增量将用最合适的开发过程来开发。在开发时，为稍后的增量准备的需求分析不断进行，但对目前增量需求的变更不会被接受。

一旦一个增量已完成并且被交付，客户就能将其派上用场。这意味着部分系统功能可以提前交付使用。开发者对系统的经验能帮助其理解后面的增量需求和目前增量后续版本的需求变更。当新的增量完成的时候，开发者将其与已存在的系统增量集成，以至于系统功能随每个所交付的增量而改进。通用性的服务可以在过程早期阶段先实现，也可以等到增量需要相应功能时再实现。

增量式开发过程的好处是：

1. 客户可以将早期的增量作为原型，从中获得对后面系统增量的需求经验。不像原型过程那样，这里的这些增量都是真正系统的一部分，因此，当系统全部完成时，用户不需要再重新学习。
2. 客户无需等到整个系统的实现才能从中获益。第一个增量会满足他们大多数关键的需求，因此，软件马上就能使用。
3. 这一过程保持了增量式开发的优点，那就是相对来讲变更能较容易地嵌入到系统中。
4. 因为具有最高优先权的服务被首先交付，而后面的增量也不断被集成进来，这就使得最重要的系统服务肯定接受了最多的测试。这也就意味着在系统的最重要的部分，客户不太可能遇到失败。

然而，增量式交付也存在一些问题：

1. 大多数系统需要一组基本设施，这些基本设施会被系统不同部分所使用。因为需求的详细定义是当增量将要被开发时才完成的，所以就很难确定全体增量所需要的公用设施。
2. 当所开发的是一个替换系统时，迭代开发也是很困难的。因为用户想要旧系统的所有功能，往往不愿意尝试一个不完整的新系统。因此，获取用户反馈是困难的。
3. 软件描述和软件本身一起开发是迭代过程的本质。然而，这与许多机构的采购模型是相冲突的，因为完整的系统描述往往是系统开发合同的一部分。在增量方法中，直到最后的增量描述完成，才会有完整的系统描述。这就需要一种新形式的合同，像政府机构这样的大客户将很难接受这样的合同。

对有些类型的系统来说，使用增量式开发和交付不是最好的方法。非常大型的系统的开发团队分布在不同的地方；一些嵌入式系统的软件取决于硬件开发；而对于要求极高的系统来说，所有的需求都需要经过分析从中查出那些可能对安全性和信息安全性造成影响的交互。

当然，这些系统和其他系统一样都存在需求的不确定性和易变性。因此，为解决这些问题并获得增量式开发的好处，可以使用一个过程，以便能迭代式开发系统原型并将其作为一个实验平台以验证系统的需求和设计。利用从原型中得到的经验，就可以对最后的需求达成一致的意见。

2.3.3 Boehm 的螺旋模型

风险驱动的软件过程框架（螺旋模型）最初由 Boehm（1988）提出来。如图 2-11 所示。它不是将软件过程用一个伴随着有从一个活动到另一个活动的回溯的活动序列来表示，而是将过程用螺旋线表示。在螺旋线中每个回路表示软件过程的一个阶段。因此，最里面的回路可能与系统可行性研究有关，下一个回路与系统需求定义有关，再下一个回路与系统设计有关，等等。螺旋模型避免了变更风险，它假定项目的结果是有风险的，并在模型中明确地包括风险管理活动以减小开发中的风险。

在螺旋线中每个回路被分成4个部分：

1. **目标设置** 为项目的这个阶段定义专门目标。指定对过程和产品的约束，而且制定详细的管理计划。分析项目风险，根据这些风险，规划可选的策略方案。

2. **风险评估和规避** 每一个项目风险确定以后要进行详细的分析，并采取措施规避这些风险。举例来说，如果有需求不合适的风险，就可能需要开发一个原型系统。

3. **开发和有效性验证** 在风险预估以后，就可以为系统选择开发模型。举例来说，如果用户界面风险是主要的，一个适当的开发模型可能是建立抛弃式原型。如果安全性风险是主要的，则基于形式化转换的开发可能是最适当的，等等。如果主要风险在于子系统集成，那么瀑布模型可能是最适当的。

4. **规划** 对项目进行评审以确定是否需要进入螺旋线的下一个环路。如果决定需要继续，就要做出项目的下个阶段的计划。

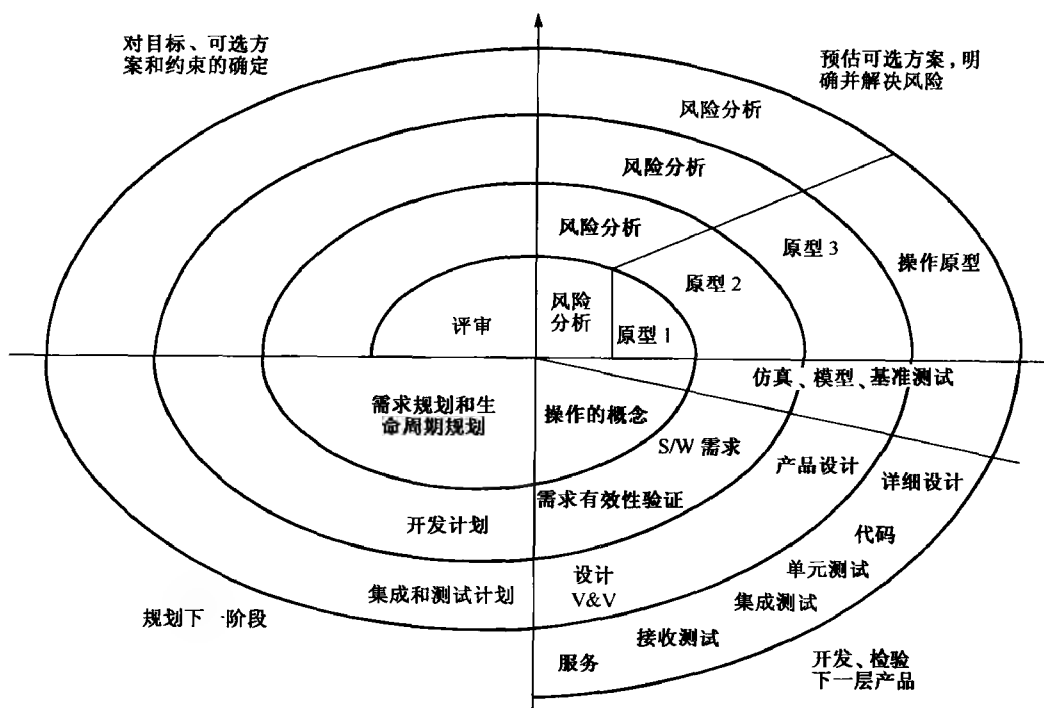


图 2-11 Boehm 的软件过程的螺旋模型 (© IEEE 1998)

螺旋模型和其他软件过程模型之间的重要区别在于，螺旋式模型中对风险的考虑是明确的。螺旋模型的一个回路始于如性能、功能等关心的目标，达到这些目标的可能方式以及对这些方式的约束在这时要全部列出。对每个目标的所有可选方案要进行评估，通常这就会帮助识别项目风险的源头。下一步骤是通过详细分析、原型建立、仿真等活动预估这些风险。

一旦风险已经得到评估，一些开发活动就可以进行了，随后就是对开发过程其他阶段的活动进行规划。通俗地说，风险就是出问题的可能性。比如，若想使用一种全新的程序设计语言，风险来自使用的编译器是否可靠或者能否产生高效率的目标代码。在项目管理中避免项目超期和超支等风险是项目管理中最重要的活动，将在第 22 章讨论这个问题。

2.4 Rational 统一过程

Rational 统一过程 (RUP) (Krutchen, 2003) 是新式过程模型的一个实例，该新式过程模型

来自于 UML 上的工作以及相关的统一软件开发过程 (Rumbaugh 等, 1999; Arlow 和 Neustadt, 2005)。由于它是一个混成过程模型的一个很好的实例, 所以在此给出一个描述。它集合了所有一般过程模型的要素 (2.1 节), 并给出了在描述和设计上的好的实践 (2.2 节), 还支持原型构造和增量交付方法 (2.3 节)。

RUP 过程认识到传统过程模型只是展现过程的一个视角。相比之下, RUP 一般从 3 个视角来描述过程:

1. 动态视角, 给出模型中随时间所经历各个阶段。
2. 静态视角, 给出所进行的过程活动。
3. 实践视角, 提出在过程中可以采用的良好实践的建议。

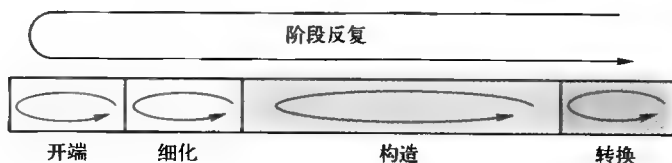


图 2-12 Rational 统一过程的各阶段

绝大多数关于 RUP 的描述都试图将静态视角和动态视角结合在一个图中 (Knutchen, 2000)。这会使得过程十分难以理解, 所以对每一个视角给出单独的描述。

RUP 是一个阶段化的模型, 识别出软件过程当中的 4 个独立阶段。然而, 不像瀑布模型各个阶段与过程活动是等同的那样, RUP 中的阶段是紧密与业务关联的, 而不是与技术层面关联的。图 2-12 给出了 RUP 中的阶段。它们是:

1. **开端** 开端阶段的目标是建立系统的一个业务案例。要识别所有与系统交互的外部实体 (人和系统) 并定义这些交互。然后使用这些信息评估系统对业务的贡献。如果这个贡献是微小的, 那么项目就要在此阶段结束时被取消了。

2. **细化** 细化阶段的目标是增进对问题域的理解, 建立系统的体系框架, 给出项目计划并识别关键项目风险。在这个阶段完成时, 就得到了系统的需求模型, 可能是一组用 UML 描述的用例、体系结构描述和开发计划。

3. **构造** 构造阶段主要关心的是系统设计、编程和测试。系统的各个部分并行开发, 然后集成在一起。在这个阶段完成时, 就得到了一个能工作的软件系统, 还有能交付给用户的相关文档。

4. **转换** RUP 的最后阶段, 关注如何将系统从开发单位转移到用户单位, 并使之在真实环境中工作。这是被绝大多数软件过程模型所忽视的东西, 而事实上这是一个代价很高且有时问题很大的活动。在此阶段完成时, 我们就有了一个在运行环境下能正常工作的文档完备的软件系统了。

RUP 中对迭代的支持有两种方式。每一个阶段可能被迭代地执行, 其结果一次次增量式地得到改善。另外, 所有阶段作为一个整体也会增量式地执行, 如图 2-12 中从转换到开端的循环箭头所示。

RUP 的静态视角聚焦在开发过程中所发生的活动上。这些在 RUP 描述中被称为工作流。在此过程中我们找出了 6 个核心过程工作流, 3 个核心支持工作流。在对 RUP 的设计中采用了 UML。UML 是一个面向对象建模语言, 所以工作流描述是围绕着相关的 UML 模型给出的, 这些 UML 模型如时序模型、对象模型等。核心工程和支持工作流的描述如图 2-13 所示。

工作流	描 述
业务建模	使用业务用例对业务过程进行建模
需求	找出与系统进行交互的参与者并开发用例完成对系统需求的建模
分析和设计	使用体系结构模型、组件模型、对象模型和时序模型来创建并记录设计模型
实现	实现系统中的组件并将它们合理组织在子系统中。从设计模型自动地生成代码有助于加快此过程
测试	测试是一个迭代过程，它的执行是与实现紧密相连的。系统测试紧随实现环节的完成
部署	创建和向用户分发产品版本并安装到工作场所
配置和变更管理	此支持性工作流用于管理对系统的变更（参见第 25 章）
项目管理	此支持性工作流用于管理系统开发（参见第 22 章和第 23 章）
环境	此工作流用于提供软件开发团队可用的合适的软件工具

图 2-13 Rational 统一过程中的静态工作流

分别表现动态视角和静态视角的好处在于，开发过程中的阶段是不与特定工作流相关联的。至少从原理上讲，各种 RUP 工作流在整个过程的所有阶段中都是可能存在的。当然，在过程的初期阶段，绝大多数工作量可能是花在如业务建模和需求这样的工作流上了，而在后期阶段中工作量主要都花在了测试和部署上了。

RUP 的实践视角描述了在系统开发中所需要的好的软件工程实践。6 个基本的且最好的实践是：

- 1. **迭代地开发软件** 根据客户的轻重缓急来规划系统的增量，在开发过程中先开发和交付最高优先权的系统特性。
- 2. **对需求的管理** 明确地记录客户的需求并跟踪这些需求的变更。在接受之前分析系统变更所带来的影响。
- 3. **使用基于组件的体系结构** 将系统体系结构组织成组件的形态，如在本章前面所讨论的那样。
- 4. **可视化地建模软件** 使用图形 UML 模型表现软件的静态和动态视图。
- 5. **检验软件质量** 保证软件满足了机构质量标准。
- 6. **控制对软件的变更** 使用变更管理系统、配置管理程序和工具来管理软件的变更。

RUP 并不是一种适合所有开发类型的过程，比如嵌入式软件开发。但是它确实代表了一种潜在地结合了 3 种基本过程模型的方法，这些基本过程模型如 2.1 节中介绍的。RUP 最为重要的创新在于把阶段和工作流相分离，以及对将软件部署到用户环境的看重。阶段是动态的而且是有目标的。工作流是静态的技术活动，且是不与单个阶段相关的但可以在整个开发过程中使用以达到每个阶段的目标的。

要点

- 软件过程是产生一个软件系统的一系列活动。软件过程模型是这些过程的抽象表示。
- 一般过程模型描述软件过程的组成。一般过程模型实例包括瀑布模型、增量式开发、面向复用的开发。
- 需求工程是开发软件描述的过程。描述的目的是向开发者传达客户方对系统的需要。
- 设计和实现过程是将需求描述转换为一个可运行的软件系统的过程。系统化的设计方法用来完成这个转换。

- 软件有效性验证是检查系统是否与它的描述相一致，以及是否符合系统用户的真正需要的过程。
- 软件进化是修改已存在的软件系统以适应用户新的需求的过程。变更是一个持续的过程，软件必须在变更过程中保持可用。
- 过程应包含应对变更的活动。这可能包含一个原型构造阶段，以帮助避免在需求和设计上的错误决定。过程应该适应迭代开发和交付，这样变更时就不会对整个系统带来干扰。
- Rational 统一过程是新式基本过程模型，其特点是由阶段（开端、细化、构造和转换）所构成，但是它把活动（需求、分析和设计等）和阶段相区别。

进一步阅读材料

《Managing Software Quality and Business Risk》这是一本软件管理方面的一本主要著作。不仅如此，这本书中也包括很棒的有关过程模型（第4章）的一章（M. Ould, John Wiley & Sons, 1999）。

《Process Models in Software Engineering》这本书对已提出的软件过程模型做了很好的概述（W. Scacchi, Encyclopaedia of Software Engineering, ed. J. J. Marciniak, John Wiley and Sons, 2001）。<http://www/ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf>。

《The Rational Unified Process—An introduction (3rd edition)》这是一本目前所能得到的关于RUP的可读性最好的一本书了。作者对过程有很好的描述，但是他本人却更看重书中有关使用此过程所面临的一些实际困难的内容（P. Krutchen, Addison -Wesley, 2003）。

练习

- 2.1 为以下各系统提出合适的软件过程模型，并基于系统所属类型给出你的理由：
 - 汽车防锁死刹车控制系统；
 - 一个支持软件维护的虚拟现实系统；
 - 大学记账系统，准备替换一个已存在的系统；
 - 一个位于火车站的交互式火车车次查询系统。
- 2.2 解释为什么增量式开发是开发商务软件系统的最有效技术？为什么这种模型不适用于实时系统工程？
- 2.3 思考图 2-3 所示的基于复用的过程模型。解释在这个过程中为什么必须要有两个分离的需求工程活动。
- 2.4 说明为什么在需求工程过程中区分用户需求开发和系统需求开发是重要的。
- 2.5 描述在软件设计过程中的主要活动以及这些活动的输出。使用图来说明在这些活动输出之间可能存在的关系。
- 2.6 解释为什么在复杂系统中，变更是不可避免的，并给出过程活动的例子（除了原型和增量交付以外的）有助于预测变更，并使已开发的软件更适应变更。
- 2.7 解释为什么采用原型开发的系统通常不被用作产品系统。
- 2.8 解释为什么 Boehm 的螺旋模型是一个适应性模型，可以同时支持变更避免和变更容忍活动。说明为什么在实践中这个模型还没有被广泛应用。
- 2.9 如在 Rational 统一过程中那样，解释提供静态视角和动态视角的优势是什么？
- 2.10 从历史角度看，技术的引进导致了前所未有的劳动力市场变化，至少临时性地取代了人们的工作。讨论大量的过程自动化的引入是否可能带来同样的结果。如果你不认为它会这样，解释为什么。如果你认为它将减少工作机会，那么受到影响的工程人员积极地或消极地抵抗这种技术是否道德？

参考书目

- Arlow, J. and Neustadt, I. (2005). *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition)*. Boston: Addison-Wesley.
- Boehm, B. and Turner, R. (2003). *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston: Addison-Wesley.
- Boehm, B. W. (1988). 'A Spiral Model of Software Development and Enhancement'. *IEEE Computer*, **21** (5), 61–72.
- Budgen, D. (2003). *Software Design (2nd Edition)*. Harlow, UK.: Addison-Wesley.
- Krutchén, P. (2003). *The Rational Unified Process—An Introduction (3rd Edition)*. Reading, MA: Addison-Wesley.
- Massol, V. and Husted, T. (2003). *JUnit in Action*. Greenwich, Conn.: Manning Publications Co.
- Rettig, M. (1994). 'Practical Programmer: Prototyping for Tiny Fingers'. *Comm. ACM*, **37** (4), 21–7.
- Royce, W. W. (1970). 'Managing the Development of Large Software Systems: Concepts and Techniques'. IEEE WESTCON, Los Angeles CA: 1–9.
- Rumbaugh, J., Jacobson, I. and Booch, G. (1999). *The Unified Software Development Process*. Reading, Mass.: Addison-Wesley.
- Schmidt, D. C. (2006). 'Model-Driven Engineering'. *IEEE Computer*, **39** (2), 25–31.
- Schneider, S. (2001). *The B Method*. Houndmills, UK: Palgrave Macmillan.
- Wordsworth, J. (1996). *Software Engineering with B*. Wokingham: Addison-Wesley.

敏捷软件开发

目标

本章的目标是介绍敏捷软件开发的几种方法。读完本章，你将了解到以下内容：

- 理解敏捷软件开发方法的基本原理、敏捷方法的核心内涵，以及它与计划驱动软件开发方法之间的差别；
- 了解极限编程的主要做法，以及它是如何贯彻和遵循敏捷软件开发方法的一般性原理的；
- 理解敏捷项目管理的 Scrum 方法；
- 了解在大型软件系统开发过程中应用伸缩的敏捷方法时的事项和问题。

当前的业务都是处于全球性、快速变化的环境中的。它们不得不对很多方面做出响应，包括新的机遇和市场、不断变化的经济条件、出现的新的竞争产品和服务。软件几乎是所有业务运行当中的一部分，所以非常重要的一点是新的软件要迅速地开发出来以抓住新的机遇，应对竞争压力。因而，快速的软件开发和交付通常已成为软件系统的最为关键性的需求了。事实上，很多业务都宁愿牺牲一些软件质量、降低某些需求来赢得快速软件交付。

因为业务运行于一个变化的环境中，实际上通常是不可能导出一个完全的和稳定的软件需求。所提出的需求不可避免地要发生改变，这是因为，客户会发现他们是不可能预知系统是如何影响他们的实际工作的，本系统是如何与其他系统进行交互的，以及用户哪些操作应该能自动完成。只有当系统交付之后且用户在有了一些系统使用经验的时候真正的需求才能浮出水面。虽然如此，由于外部因素的影响，需求的变更可能来得非常快，且是不可预知的。这样软件就可能延期交付。

建立在对需求描述，然后进行设计，构造，最后再进行测试的完整计划上的软件开发过程是不适应快速软件开发的。当需求发生改变，或者是当需求出现问题时，系统设计和实现不得不返工和重新进行测试。其结果是，传统的瀑布模型或基于描述的过程总是拖延，最后的软件交付给客户的时间远远晚于最初的规定。

对有些类型的系统，如安全要求极高的控制系统，对其完整的系统分析是十分重要的，采用计划驱动的方法是恰当的。然而，在快速变化的业务环境中，这能引发实际的问题。在软件投入使用的时候，最初购买该软件的理由可能已经完全不复存在了，所以实际上软件此时已经变得毫无用处。因此，尤其是对业务系统，瞄准快速软件开发和交付的开发过程就十分关键了。

人们认识到对快速的系统开发和能够处理不断变更的需求的过程的需要已经有一段时间了。IBM 在 20 世纪 80 年代介绍了增量开发（Mills 等，1980）。也是在 20 世纪 80 年代，第四代语言的引入支持快速开发和交付的思想（Martin，1981）。然而，这个概念的真正得到发展是在 20 世纪 90 年代后期，伴随着敏捷方法的发展，如 DSDM（Stapleton，1997）、Scrum（Schwaber 和 Beedle，2001）、极限编程（Beck，1999；Beck，2000）。

快速的软件开发过程就是为迅速制造可用软件而设计的。软件的开发和部署不是一次完成的，而是以一系列增量的形式完成的，每一个增量包括新的系统功能。尽管有很多快速软件开发的方法，它们都有一些基本的特性：

1. 描述、设计和实现过程是交织在一起的。没有详细的系统描述，设计文档得到了最少化，

或者是由实现系统所采用的编程环境所自动生成。用户需求文档只定义最为重要的系统特性。

2. 系统通过一系列版本开发出来。最终用户和其他系统信息持有者都参与了每个版本的定义和评估。他们提出对软件的变更建议和对系统后一个版本应该实现的新需求的提议。

3. 系统用户界面通常是采用交互式开发系统开发的, 这些开发系统允许通过绘图和在界面上摆放图标的方式迅速完成界面的设计。系统可以为浏览器生成基于 Web 的界面, 或者是为专门平台比如微软视窗设计一个界面。

敏捷方法是增量式开发方法, 每个增量一般都比较小, 通常, 每两三个星期就会创建系统的新版本并提供给用户使用。用户参与开发过程, 以便快速获得变更需求的反馈。利用非正式的沟通、而不是采用有书面文件的正式会议, 这样使文档制作量最少化。

3.1 敏捷方法

在 20 世纪 80 年代和 90 年代初, 人们对于取得好的软件的最好方法的普遍共识是: 需要通过仔细的项目规划和形式化质量保证, 采用 CASE 工具所支持的分析和设计方法, 遵循受控的和严格的软件开发过程。这些观点来自于软件工程领域中关注大型、长生命期且是由大量单体程序所构成的软件系统的开发的那些人。

这种软件是由大型团队所开发的。通常他们是地理上是分散的, 而且所进行的开发工作时间持续很长。这类软件的例子是现代飞机控制系统, 可能需要 10 年左右的时间完成从初始描述直到部署完毕。这些基于计划的方法涉及在规划、设计和文档生成方面的高额的费用。这个费用在下列情况下是合理的, 这些情况包括: 当多个开发团队必须协同工作、所开发的系统是要求极高的系统, 或者是当很多人力需要投入到软件的整个生命期的维护工作中去的情形。

然而, 当这个重量级的、基于计划的开发方法应用于小型或者是中等规模业务系统时, 所需要的费用在软件开发过程中所占的比例非常大, 以至于主宰整个开发过程。更多的时间花在了系统应该如何开发而不是花在程序开发和测试上。当系统需求发生了变更, 返工就是很严重的问题了, 至少在理论上讲, 描述和设计都是需要随着程序的改变而改变的。

对这种重量级的方法不满引发了众多的软件开发人员在 20 世纪 90 年代提出了新的敏捷开发方法。这些方法允许开发团队将主要精力集中在软件本身而不是在设计和编制文档上。敏捷方法普遍地依赖于迭代方法来完成软件描述、开发和交付。最适合系统需求在开发过程中快速变化的应用类型。软件开发人员打算用它们来迅速完成和交付可用软件给客户, 客户提出新的变化的需求, 由软件开发人员在下一个循环中实现。他们的目标是减少开发过程中的繁琐多余的部分, 通过避免那些从长远看未必有用的工作和减少可能永远都不会被用到的文档的方法达此目的。

敏捷方法背后的基本原理体现在敏捷宣言中。此宣言由那些创建这些方法的主要人物所提出。宣言如下:

我们在开发过程中发现更好的软件开发方法, 并帮助他人这样做。通过这项工作我们得到如下评估:

个体和交互胜过过程和工具;

编写软件胜过书写详尽的文档;

用户合作胜过合同谈判;

响应变更胜过遵循计划;

也就是说, 虽然右边的项有价值, 但我们更重视左边的项的价值。

也许最为有名的敏捷方法就数极限编程了 (Beck, 1999; Beck, 2000), 下一小节将专门介绍这个主题。其他敏捷方法还包括 Scrum (Cohn, 2009; Schwaber, 2004; Schwaber 和 Beedle,

2001), Crystal (Cockburn, 2001; Cockburn, 2004), 适应性软件开发 (Highsmith, 2000), DS-DM (Stapleton, 1997; Stapleton, 2003) 和特征驱动开发 (Palmer 和 Felsing, 2002)。这些方法的成功还导致了它们与更传统的开发方法的集成, 这种建立在系统建模基础上的集成方法形成了所谓的敏捷建模的概念 (Ambler 和 Jeffries, 2002), 以及 Rational 统一过程的敏捷实例化的概念 (Larman, 2002)。

尽管这些敏捷方法都是建立在增量式开发和交付的概念上的, 但达到这样的目标所用的过程是不同的。然而, 它们的基本原则是相同的, 即都基于敏捷宣言, 因而有很多共同点。这些基本原则如图 3-1 所示。不同的敏捷方法用不同的方法实例化这些准则, 这里就不一一赘述。在此主要介绍两个最为广泛应用的方法: 极限编程 (见 3.3 节) 和 Scrum (见 3.4 节)。

原 则	描 述
客户参与	客户应该在开发过程中始终紧密参与其中。他们的作用是提供新系统的需求、对需求排序, 并评估系统的迭代
增量式交付	软件以增量的方式进行开发, 客户指定在每个增量中将要包含的需求
人非过程	开发团队的技术应该得到承认和发扬, 团队成员应该保持他们自己的工作风格, 不落俗套
接受变更	预料系统需求的变更, 并设计系统使之适应这些变更
保持简单性	致力于所开发的软件 and 开发过程的简单性。只要可能, 就积极地去排除系统中的复杂性

图 3-1 敏捷方法的基本原则

对下面几个类型的系统开发, 敏捷方法是非常成功的:

1. 软件公司正在开发并准备出售的是一个小型或中型的产品。
2. 机构内部的定制系统的开发。客户会有一个关于参与到开发过程中去的明确承诺, 且没有许多来自外部的规章和法律等影响。

正如本章最后一节将介绍的, 敏捷方法的成功意味着人们有极大的兴趣要将这些方法应用到其他类型的软件开发中。然而, 因为它致力于小的、紧密集合的团队, 将它们扩展到大型系统中就会有许多问题。但也已经有安全要求极高的系统工程中应用敏捷方法的经验 (Drobna 等, 2004)。尽管如此, 因为要求极高的系统对信息安全性、安全性、可靠性分析的需要, 敏捷方法在应用到要求极高的系统工程中之前, 需要进行重大修改。

在实践中, 敏捷方法所基于的基本原则有时是很难付诸实施的:

1. 虽然让客户参与到开发过程中来的想法是很吸引人的, 但是它的成功依赖于有愿意加入进来且肯在与开发团队的沟通上花时间的人, 而且此人还要能够代表所有的信息持有者。常发生的情况是, 客户代表屈从于其他压力而不能够全身心地投入到软件开发中来。
2. 团队成员可能从性格上不太适应大强度的投入, 而这又正是敏捷方法的典型特征。因而他们可能不能够做到与其他成员的良好沟通。
3. 对变更做出优先级排序可能是极其困难的, 尤其是对那些拥有很多信息持有者的系统。典型情形是, 每个信息持有者会给出一个不同的优先级排序。
4. 维护简单性需要额外的工作。迫于交付时间表的压力, 团队成员会没有时间执行应该有的系统简化过程。
5. 许多机构, 特别是大公司, 已花费数年致力于改变他们的文化, 以保证制定和遵循过程规范。他们很难转向另一种工作模型, 这种工作模型中的过程是非正规的, 而且是由开发团队制定的。

另外还会出现一个非技术性问题, 这个问题也是增量式开发和交付的一般问题, 即在系统

客户通过某外部机构进行系统开发时所发生的问题。软件需求文档总是客户和提供商之间合同的一部分内容。因为增量描述是敏捷方法的固有内容，为此类开发写合同是件困难的事情。

因此，敏捷方法不得不依赖这样的合同，即客户根据系统开发所需要的时间来支付费用，而不是根据需求描述来支付费用。如果一切发展顺利，这对客户和开发者来说都将是有利的。然而，如果问题出现了，那么双方就会相互指责，对谁该负责为解决问题所花费的额外时间和资源费用发生争吵。

大多数介绍敏捷方法和经验的书籍和论文都是讨论利用这些技术进行新的系统开发。但是，正如第9章所解释的，大量的软件工程努力维护和进化现有的软件系统。仅有一小部分的经验报告是关于使用敏捷方法进行软件维护的（Poole 和 Huisman, 2001）。在思考敏捷方法和维护的时候，需要考虑两个问题：

1. 由于开发过程中强调正式文档的最小化，用敏捷技术开发的系统是否可维护？
2. 敏捷方法是否能有效地用于进化系统以响应用户的变更请求？

正式的文档应该是描述系统的，使用户更容易理解系统。然而，在实践中，正式的文档经常不能及时更新，因此就不能正确地反应程序源码。因为这个原因，敏捷方法的推崇者们争辩说，写文档是浪费时间，实施可维护的软件关键是编写高质量的、可读的代码。因此，敏捷实践强调编写结构良好代码的重要性，并致力于改善代码。所以，用敏捷方法开发的系统，维护时文档的缺乏不应该成为一个问题。

但是，在系统维护上，作者的经验是：关键文档是系统需求文档，它告诉软件工程师们软件系统应该做什么。如果没有这些信息，很难评估所建议的系统变更的影响。许多敏捷方法非正式的、增量式地收集需求，也不建立有条理的需求文档。为此，使用敏捷方法很可能使随后的系统维护变得更难、更昂贵。

不管系统开发时是否用了敏捷技术，敏捷实践用在维护过程本身可能是有效的。增量式交付、面向变更和维护简单化的设计对于变更的软件是有很大大意义的。实际上，可以认为敏捷开发过程是一种软件进化的过程。

然而，软件交付后的主要困难可能是继续让用户参与到过程中。尽管在系统开发期间，客户会认为派代表全职参与是正当的，但在维护期间可能就不会这么认为，全职参与的可能性极小，因为他们会认为维护期间的变更是不连续存在的。用户代表们可能会对系统丧失兴趣。因此，这时可能就需要找到一种替代机制来创建新的系统需求，像第25章介绍的变更建议。

可能出现的另一个问题是保持开发团队的持续性。敏捷方法依赖团队成员理解系统的各个方面，因为没有参考文档可循。如果一个敏捷开发团队解散了，这种隐性的知识就丢失了，这样，一个新的团队很难对系统及其组件有相同的理解。

敏捷方法的支持者们热衷于促进对这些方法的使用而忽视了它们的缺点。这也引发了同样极端的响应，从作者个人观点看，这就是夸大了该方法所存在的问题（Stephens 和 Rosenberg, 2003）。更理性的批评可见 DeMarco 和 Boehm（DeMarco 和 Boehm, 2002），突出了敏捷方法的优点和缺点。他们提出一种混成方法，即敏捷方法嵌入来自计划驱动式开发的某些技术，这将是今后最为有效的做法。

3.2 计划驱动开发和敏捷开发

软件开发的敏捷方法认为设计和实现是软件过程的核心活动。敏捷方法将其他的活动，如需求的导出和测试，合并到设计和实现活动中。相对而言，软件工程的计划驱动方法，识别软件过程中的每个阶段及其相关输出。前一个阶段的输出作为规划接下来的过程活动的基础。图3-2所示的是，对于系统描述，计划驱动和敏捷方法之间的不同。

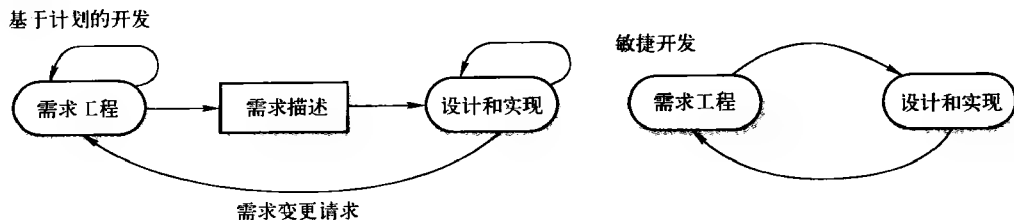


图 3-2 计划驱动和敏捷描述

在计划驱动的方法中，迭代发生在各个活动之中，用正式文件在软件过程的各个阶段之间进行沟通。例如，需求将演化，最终，将产生一个需求描述。这又作为设计和实现过程的输入信息。在敏捷方法中，迭代发生在所有活动之间。因此需求和设计是一起开发的，而不是单独进行的。

计划驱动的软件过程可以支持增量式开发和交付。分配需求并将设计和开发阶段计划为一系列的增量是完全可行的。敏捷过程并非一定是围绕代码这个焦点的，它也可以产生一些设计文档。正如在下节中介绍的，敏捷开发团队可能决定去完成一个文档的‘spike’，团队生产出一个系统文档，而不是生产出一个新版本系统。实际上，大多数的软件工程都包括计划驱动的开发和敏捷开发的实践。为了在计划驱动和敏捷方法之间得到平衡，你必须回答以下一些技术的、人员的和机构方面的问题：

1. 在实现开始之前，有非常详细的描述和设计很重要么？假如是，你可能需要用计划驱动的方法。
2. 增量交付策略，即软件交付给用户并快速地取得反馈，切实么？假如是，考虑使用敏捷方法。
3. 开发的系统有多大？敏捷方法对于小的、处于同一地点的开发团队来说大多是有效的，这种团队的交流往往是非正式的。可能不适于需要大的开发团队的大型系统，这种系统可能要用计划驱动的方法。
4. 开发的系统类型是什么？实施之前需要大量的分析的系统（如有复杂时序需求的实时系统），通常需要相当详细的设计来实现这些分析。这种情况下，计划驱动的方法可能是最好的。
5. 预想的系统寿命是多长？长寿命的系统可能需要通过更多的设计文档来交流系统开发者的最初意向，以支持团队工作。然而，敏捷方法的支持者们说文档通常不能及时更新，且在长期的系统维护中不经常用到。
6. 有什么样的技术来支持系统开发？敏捷方法通常依赖于好的工具，以跟踪设计进化。假如你于系统开发中使用 IDE，而又没有好的可视化编程和开发工具，就可能需要更多的设计文档。
7. 开发团队是怎么组织的？如果开发团队是分散的或一部分的开发是外包的，你可能就需要开发设计文档，以在开发团队之间进行沟通。你可能就需要提前做计划。
8. 有没有可能影响系统开发的文化问题？传统的工程机构有计划驱动开发的文化，这是一个工程规范。这通常需要额外的设计文档，而不是在敏捷过程中非正式的信息。
9. 开发团队的设计人员和编码人员的能力如何？有时敏捷方法需要有比计划驱动的方法更高的技术水平，计划驱动的方法中编程人员可以简单地将一个详细的设计转化成代码。假如你的团队水平相对较低，你可能需要用最好的人员设计开发，其他人负责编程。
10. 系统是否受制于外部的法规？假如系统有其他外部的法规限制（如联邦航空管理局 FAA 核准一个安全性要求极高的航空操作软件），你可能需要有详细的文档，作为系统安全案例的一

部分。

实际上，一个项目是否可以被划为计划驱动的或敏捷开发的问题并不是很重要。最终，软件系统的购买者主要关注的是这个可执行软件是否能满足他们的需求，是否对个人用户或机构有用。在实践中，许多声称使用了敏捷方法的公司接受了某些敏捷实践，并将它们和计划驱动的过程集成在一起。

3.3 极限编程

极限编程（XP）也许是最为熟知也是流行最广的一种敏捷方法。其名字是由 Beck（Beck，2000）所创造的，因为该方法是推行公认的好的经验，例如迭代式开发，到一个“极限”水平。比如，在 XP 中，系统的多个新版本可能由不同的程序员实现并集成和测试，在一天内完成。

在极限编程中，所有的需求都表示为脚本[⊖]（称为用户故事情节），它将被直接实现为一系列任务。程序员两两配对工作，在写代码之前为每个任务开发测试。在新的代码加入到系统中时，所有的测试必须成功执行。图 3-3 说明了一个 XP 过程，它产生了正在开发的系统的一个增量。

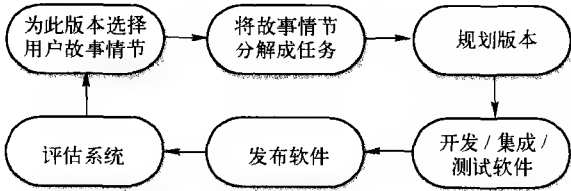


图 3-3 极限编程的版本循环

极限编程包含多个实践，图 3-4 给出了一个摘要，这些反映了敏捷方法的一些原则：

1. 增量式开发是通过系统小的频繁的版本来支持的，其间所采用的对需求描述的方法是基于客户故事情节或脚本，这样的故事情节或脚本作为决策哪些功能需要放在增量里的根据。
2. 客户的参与是采用全时雇佣到开发团队的方式。客户代表参与开发并负责定义系统的接收测试。
3. 人，而不是过程，是通过结对编程、对系统代码的集体拥有、可持续的开发过程而无需超额的工作时间来运作的。
4. 对变更的支持是通过经常性的系统版本发布、测试优先开发、重构以避免代码退化以及连续的集成新功能等方法实现的。
5. 对简单性的维护是通过持续的重构来改善代码质量、使用简单的设计以减少系统将来的变更等方式来支持的。

在 XP 过程中，客户亲密地投入到系统需求的定义和优先权排序工作中。需求不是定义为所要求的系统功能的列表。系统客户是开发团队的一分子，与其他团队成员一起讨论脚本。同时，他们用一个所谓的“脚本卡”封装客户需要。然后开发团队瞄准这个目标在软件的将来版本中实现这个脚本。心理健康病人管理系统的脚本卡例子如图 3-5 所示，这是给病人开处方药的场景的简短描述。

⊖ 这里“脚本”对应的英文单词是 scenario，有些译者将该词译为“场景”或“情景”，本译稿中根据上下文语境三种译法都有采用。——译者注

原则或实践	描 述
增量式规划	需求记录在脚本卡片上，包含在版本中的故事情节可以决定可用的时间和它们的相对优先级。开发者将这些脚本分解为开发的“任务”。参见图 3-5 和图 3-6
小版本发布	首先开发能提供业务价值的一个最小有用集合。不断地增量式地往第一个版本中添加功能形成新的系统版本
简单设计	只进行有限的能满足当前需求的设计，不追求太多
测试优先的开发	在功能本身实现之前，采用一个自动单元测试框架来编写对此新功能的测试
重构	期待所有的开发人员都能连续地对代码重构，只要是发现有可能改善的代码就去做，这样能保持代码简单性和可维护性
结对编程	开发人员成对工作，检查彼此的工作并提供支持圆满完成任务
集体所有	配对的开发人员参与系统的所有方面的工作，所以不会存在技术孤岛，所有的开发者都享有代码。任何人都可以修改任何地方
连续集成	任务一完成，将它集成到大系统中。在每次这样的集成后，必须通过系统中所有的单元测试
可持续的节奏	大量超时是不能接受的，因为它的后果通常就是降低代码质量和平均生产率
在场客户	系统最终用户的代表（客户）应该全程配合 XP 团队。在极限编程过程中，客户是开发团队的一分子，有责任将系统需求带给开发团队

图 3-4 极限编程实践

开 处 方

凯特是一位医生，她要为来诊所的病人开处方药。病人记录已经显示在她的电脑上了，因此她点击药物栏可以选择“当前药物”、“新药”或“药典”。

假如她选择“当前药物”，系统会询问她药量。假如她想更改，就要键入确认，然后确认处方。

假如她选择“新药”，系统假设她知道要开那种药。她要键入药品名称的首字母。系统就会显示可能的药品列表。她选择需要的药品，系统响应会询问她选择的是否正确。她键入确认并确认处方。

假如她选择“药典”，系统会显示搜索框，搜索已存在的药典。她可以搜索需要的药品。她选择一种药品，系统会询问她选择的是否正确。她键入确认并确认处方。

系统通常会确认剂量是否在允许范围内。假如不是，凯特就会改变已做的。

凯特确认处方之后，处方将显示出来以便确认。她可以点击“确认”或“变更”。假如她点击“确认”，处方将会被记录在已审计数据库中。假如点击“变更”，她将重新键入“开处方”过程。

图 3-5 开处方的脚本

脚本卡是 XP 规划过程或“规划游戏”的主要输入。一旦脚本卡做出来，开发团队就把每个脚本拆分成任务（见图 3-6）并估计实现时所需要的人力和资源。这通常要与客户讨论，重新定义需求。客户然后对脚本进行优先权排序以便实现，选择那些马上能使用到的脚本尽快交付可用的业务支持。意图是识别出那些可在两周内实现的有用的一些功能特性，当下一个版本发布时即可被用户使用。

当然，在需求发生变化的时候，未实现的脚本发生改变或者被忽视掉。如果需要对已经交付的系统进行变更，要制作出新的脚本卡，而且客户需要重新决定是否这些改变应该有比新功能更高的优先级。

有时，在规划过程中，暴露出的问题是不好解决的，需要额外的工作探求可能的解决方案。该团队可进行一些原型或尝试性开发以了解问题和解决策略。在 XP 术语中，这是一个 spike，是一个无需编程的增量。可能也有设计系统构架或开发系统文档的 spikes。

极限编程将增量式开发推向极致。新的软件版本一天之中要构造好多次，移交给客户的增量大约是每两周一次。从不拖延发布的截止日期；如果有开发问题，将咨询顾客，并将功能从计

划发布的版本中移除。

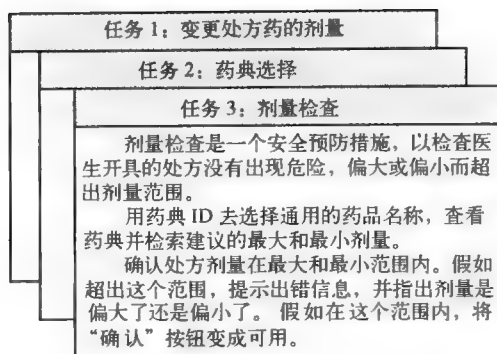


图 3-6 开处方的任务卡

当程序员创建出新的版本, 他/她必须运行所有现存的自动化测试以及对新功能的测试。只有当所有测试都成功执行之后软件的新版本才是可接受的。这将成为系统下一个迭代的基础。

传统软件工程的基本训诫是你必须为变更而设计。也就是说, 你应该预见到软件未来的变更, 面向变更地设计软件以便这些变更能很容易地实现。然而, 极限编程抛弃这个做法, 这是基于这样一种认识, 即为变更而设计通常是很浪费人力的。所预见的变更通常不能变成事实, 而完全不同的变更请求却总是层出不穷。因此, XP 接受即将发生的变更, 并在变更确实发生时重组软件。

增量开发的一个普遍问题是, 它有使软件结构变坏的趋势, 所以变更越来越难以实现。本质上, 程序员总是在想方设法找到一个变通方法以解决问题, 代码经常重复, 部分软件不恰当地使用, 代码添加到系统中使整体结构变坏。

极限编程对此问题的解决是通过对软件的不不断重构。这意味着编程团队对软件查找可能的改善并马上实现它。当一个团队成员发现代码可改进, 即使在没有紧急需要的情况下, 他们也会改进它。重构的例子包括: 类的层次结构重组以消除重复代码, 整理和重命名属性和方法, 调用程序库中的方法替换代码。程序开发环境, 比如 Eclipse (Carlson, 2005), 包括重构的工具, 可以简化寻找代码不同部分间的依存关系并进行全局代码修改。

原则上, 当新的脚本卡实现时软件应该总是做到容易被理解且容易进行变更。但在实践中, 这通常不成立。有时, 开发压力使得程序不能马上进行重构, 因为大家需要马上完成某些新功能。很多新特性和变更不能够及时通过代码级的重构反映出来, 这就需要系统架构进行修改。

在实践中, 许多采用 XP 的公司并没有采用所有的图 3-4 列出的极限编程实践。他们只选择适合他们自己的工作方法。比如, 有些公司发现结对编程是有用的, 其他的公司则喜欢个人编程, 然后检查。为了适应不同的技术水平, 一些程序员对系统中不是他们开发的部分不进行重构, 他们会使用常规的需求, 而非用户的脚本卡。

3.3.1 极限编程中的测试

如本章引言部分所讨论过的, 在增量开发和计划驱动的开发之间的重要区别就在于系统测试的方式不同。在增量开发中, 没有系统描述可以被外部的测试团队用来制作系统测试。由于这个原因, 和计划驱动的测试相比, 很多增量开发方法的测试过程是很不规范的。

为了避免一些测试和系统验证方面的问题, XP 方法更加强调测试过程。系统测试是 XP 方法的核心内容, 为此引入了一个方法来降低由于新系统增量而带入系统新的错误的的可能性。

XP 当中的测试的关键特性如下：

- 1. 测试优先的开发；
- 2. 来自脚本的增量式测试开发；
- 3. 用户参与测试开发和有效性验证；
- 4. 自动测试系统的使用。

测试优先的开发是 XP 的最为重要的创新之一。不同于先写代码，然后为这些代码写相应的测试程序，XP 是先写测试程序，再写代码的。这就意味着，在写程序的同时你可以运行测试代码，以便在开发过程中及时发现问题。

先写测试隐含地定义了界面和要开发的功能的行为描述。减少了对需求和界面的误解。对于任何在系统需求和实现该需求的代码之间有明确关系的过程中都可以采用此方法。在 XP 中，你总可以看到这个关联，这是因为表述需求的脚本卡被分解为一些任务，而任务是实现的主要单元。在 XP 中采用测试优先的开发使更多测试驱动一般技术应用于开发（Astels，2003）。第 8 章将介绍这些主题。

在测试优先开发中，实现任务的人必须彻底理解描述，这样他们才能为系统编写测试。这意味着在描述中的二义性和遗漏必须在实现开始之前得以澄清。进一步，它还避免所谓的“test - lag”问题。这可能导致系统开发人员工作进度快于测试人员，使得实现比测试越来越超前，产生了一种跳过测试过程以保持进度要求的倾向，使得开发日程表不变。

在 XP 中的用户需求表达为脚本或故事情节，用户再对需求进行优先权排序以进行开发。开发团队评估每一个脚本并将其分解为任务。例如，处方药（见图 3-5）的脚本卡所导出的任务卡如图 3-6 所示。每一个任务生成一个或多个单元测试来检查此任务中所描述的实现。例如，图 3-7 是一个测试用例的简短描述，该用例是用来检查药物剂量是否超出了安全范围。

测试 4：剂量检测
<p>输入：</p> <ul style="list-style-type: none">1. 以 mg 为单位的一个数，表示单次服药剂量。2. 一个数，表示每天服药的次数。 <p>测试：</p> <ul style="list-style-type: none">1. 输入测试，这里的单次服药剂量是正确的，但服药的次数太多。2. 输入测试，这里单次服药剂量太大或太小。3. 输入测试，这里单次服药剂量 × 服药次数太大或太小。4. 输入测试，这里单次服药剂量 × 服药次数在允许范围内。 <p>输出：</p> <p>显示正确或错误信息，指示剂量在正常范围或超出安全范围。</p>

图 3-7 剂量检测的测试用例描述

在测试过程中用户的作用是：将要在系统的下一个版本中实现的故事情节开发接收测试。如第 8 章中所讨论的那样，接收测试是系统用客户数据来进行测试、看它是否满足了客户的真正需要的过程。

在 XP 中，接收测试就像开发一样，是一个增量式的。当开发继续进行时，作为团队一部分的用户进行测试。因此，所有的代码都是经过验证的，确保是满足顾客需要的。图 3-5 中的脚本卡，接收测试将涉及以下情景：a) 改变药物的剂量，b) 选择新药，c) 使用药典去发现某一药物。在实践中，通常需要进行一系列的接收测试，而不是单个测试。

依赖于客户来支持接收测试的模式有时是 XP 测试过程中的主要困难。接受客户角色的人可

能只有非常有限的时间，不大可能以全时方式与开发团队一起工作。客户可能感觉提供了需求已经尽到了义务，不太情愿投入到测试过程当中。

测试优先的开发和自动测试系统的使用是 XP 方法的主要优势。在任务实现之前就将测试写成了可执行的组件。这个测试组件应该是独立的，应该模拟待测试的输入提交，而且应该检查结果满足了输出描述。自动测试框架是这样的一个系统，它使编写可执行测试程序和提交测试集变得很容易。JUnit (Massol 和 Husted, 2003) 就是一个被广泛应用的自动测试框架的例子。

由于测试是自动的，我们就总会会有一个能快速且容易执行的测试集合。这意味着无论什么时候有功能添加进来，测试都可以执行，由新代码所引入的问题能够马上被查出来。

测试优先的开发和自动测试通常导致要编写和执行大量的测试程序。然而，这种方法并不一定使程序得到彻底的测试。有以下 3 个原因：

1. 程序员更喜欢编程而不是测试，有时在写测试时会走捷径。例如，写出不完整的测试无法检查所有可能的异常情况。
2. 有一些测试是非常难写的。例如，在复杂用户界面中，通常编写用于实现“显示逻辑”和屏幕间工作流的单元测试是十分困难的。
3. 对一组测试的完整性的判断也是困难的。尽管你可以有很多系统测试，你的测试集合可能不能提供完整的覆盖。系统的关键部分可能得不到执行，所以也就是没有经过测试的。

因此，虽然大量且频繁的执行测试集可能会让你觉得系统是完整和正确的，但事实未必如此。如果开发完成后没有进行测试复查和进一步地编写测试程序，则未被发现的错误将可能在系统交付中出现。

3.3.2 结对编程

XP 中另一个有创新性的实践是程序员结对开发软件。他们坐在同一个工作台前开发软件。但并不总是同一对程序员在一起工作，而是会轮流交替地两两结对编程，这种结对是动态变化的，这样整个团队成员就有机会在整个开发过程中与其他成员一起工作。

结对编程有下列一些优点：

1. 它支持共同拥有软件 and 共同对系统负责。这反映了 Weinberg 的非自我的程序设计理念 (Weinberg, 1971)，即软件完全由团队共同所有，个人不对代码中所出现的问题负责。相反，团队要对解决这些问题全体负责。
2. 它担当了非正式的复查过程，因为每一行代码至少经过了两个人的批阅。代码检查和复查（在第 24 章中介绍）对于发现绝大多数软件错误是非常成功的。然而，结对编程的磨合是需要时间的，特别是会带来开发过程的延长。结对编程是一种不太正规的过程，无法发现太多的错误，对比正规程序检查过程来说它是一种更加便宜的检查过程。
3. 它有助于支持重构，这是一个软件改善的过程。在正规开发环境中实现结对编程的困难在于这种投入是一种长期回报，对于单个成员来说，执行重构任务的人会被视为比只负责进行编程的程序员效率要低。而结对编程和集体所有的做法，使得其他成员马上就能从重构中受益，所以他们比较乐意支持这个过程。

有人可能认为结对编程效率会比较低，也许只有单独编程时的一半。而研究表明，事实并非如此。结对编程的生产率与单个人编程的生产率是相当的 (CockBurn 和 Williams, 2001; Williams 等, 2000)。其原因是两个人在开发之前对软件的讨论会减少错误的发生和减少返工，由非正规的检查所避免的错误数量减少了因在测试阶段发现缺陷而进行修改的时间。

然而，对更有经验的程序员 (Arisholm 等, 2007; Parrish 等, 2004) 的研究结果却并不是这样。他们发现，与两个程序员单独工作相比，生产力损失了很多。结对编程是有一些质量效益，

但却不足以抵消它的开销。但信息的共享在结对编程时是很重要的，因为当有团队成员离开时，它降低了项目的整体风险。就其本身来说，这可能就是结对编程的价值所在。

3.4 敏捷项目管理

项目经理的基本职责是管理项目，以确保软件在预算范围内按时交付。他们监督软件工程师的工作，并检测软件开发的进展。

项目的标准方法是计划驱动的。如第23章中将介绍的，经理制定一个计划，包括该交付什么，什么时候交付及谁来完成交付。一个计划驱动的方法要求经理对于每件事情都有一个稳定的看法，包括必须开发什么和开发进度。但是，这不适于敏捷方法，敏捷方法的需求开发是增量式的，短期内快速地交付增量，经常有需求和软件的变更。

像其他专业软件开发过程一样，对敏捷开发必须加以管理，以便更好地利用时间和对团队有价值的资源。这需要一种适合于增量开发（特别是敏捷方法的）、不一样的管理方法。

Scrum 方法（Schwaber, 2004; Schwaber 和 Beedle, 2001）是一个通用的敏捷方法，但它主要是注重迭代开发的管理，而不是管理敏捷软件工程的专门的技术方法。图3-8是Scrum的管理过程图。Scrum没有规定要使用的编程实践方法，如配对编程和测试优先的开发。因此，它可用于更多的敏捷方法如XP，它提供了一个项目管理的框架。

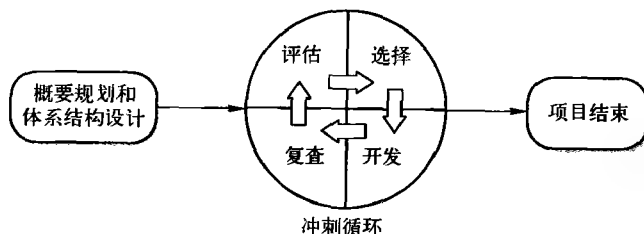


图3-8 Scrum 过程

Scrum 有3个阶段。第一个是规划纲要阶段，建立大致的项目目标和设计软件体系结构。接下来是一系列的冲刺循环，每个循环开发出一个系统增量。最后，项目结束阶段总结项目，完善需要的文档，如系统帮助和用户手册，并总结从项目中学到的经验。

Scrum 的创新在于它的中心阶段，也就是冲刺循环。一个循环是一个计划单元，需要做的工作有：评估，特征的选择和开发，软件实现。循环的最后一个阶段，所开发的全部功能都将交付给信息持有者。这个过程的主要特点如下：

1. 冲刺有一个固定的长度，一般是2~4周。在XP中，对应于一个系统版本的开发。
2. 规划的起点是所谓的 backlog（积压的任务），是项目中要完成的工作清单。在评估阶段，这些积压的任务经过审查，对它们进行优先级排序并进行风险的指派。此过程中，用户都紧密地参与，在每一个循环开始时提出新的需求或任务的建议。
3. 在选择阶段，项目团队的所有人员都要参加，和用户一起选择在冲刺循环中要开发的特性和功能。
4. 一旦这些都得到同意，团队将组织进行软件开发。每天，团队的开发成员都要参加短时会会议，回顾开发过程，若有必要，将重新安排工作。这个阶段，开发团队是隔离于客户和机构的，所有的交流都通过所谓的“Scrum master”进行。Scrum master 使开发团队免受外界干扰。工作方式取决于遇到的问题和团队。不像XP，Scrum 对于如何写需求、测试优先开发等不做具体要求。但是，如果团队觉得合适，这些XP的实践也可以用。

5. 冲刺循环的结束, 对已做工作复查并交付给用户。接着, 下一个循环就开始了。

Scrum 背后的思想是整个团队被赋予决定的权利, 因此术语“项目经理”被故意取消了。“Scrum master”是一个调解人, 安排每天的会议、跟踪日志上要做的工作、记录决定、衡量工作进展情况、与客户和团队外的管理者进行沟通。

整个团队必须出席每天的会议, 有时是“站立”着开会的, 以使其时间短且重点明确。会上, 团队成员们共享信息, 说明工作进度, 遇到的问题及接下来一天的工作计划。这就意味着每一个团队成员都知道接下来干什么, 假如出现问题, 也可以重新做短期工作计划。每个人参与短期计划制订, 而不是 Scrum master 自上而下发指令。

在 Web 上有许多 Scrum 成功运用的趣闻。Rising 和 Janoff (2000) 介绍了其在电信软件开发环境中成功使用的例子, 并列其优点如下:

1. 产品被分解成一组可管理的和可被理解的块。
2. 不稳定的需求并不阻碍工程进度。
3. 整个团队的所有事情都是可见的, 因此改善了团队的沟通。
4. 用户看到增量的及时交付, 且得到对于产品如何工作的反馈。
5. 客户和开发者之间彼此信任, 创造了积极的文化, 每个人都希望项目成功。

最初设计时, Scrum 是为设在同一地点的团队使用的, 所有的团队成员每天都一起参加“站立”会议。但是, 许多软件开发的团队分布在世界的不同地方。因此, 为使 Scrum 适应分布式的开发环境, 人们进行了各种各样的实验 (Smits 和 Pshigoda, 2007; Sutherland 等, 2007)。

3.5 可扩展的敏捷方法

敏捷方法的开发是为在同一个房间办公与交流的小开发团队使用的。因此敏捷常被用于中小型系统的开发。当然, 快速发布软件, 使之更贴近用户的需求的方法, 也适用于大型系统。所以, 现在人们对伸缩的敏捷方法 (使之去适应大部门开发大型系统), 产生了极大的兴趣。

Denning 等 (2008) 认为避免软件工程中不符合客户需求、预算超支等常见问题的唯一途径就是将敏捷方法运用于大型系统。Leffingwell (2007) 讨论将敏捷实践放大到大型系统开发中。Moore 和 Spens (2008) 报告了他们使用敏捷方法去开发一个大型医疗系统的经验, 其中有 300 个开发人员, 而且是分部在不同地域。

大型软件系统开发和小型系统开发有许多不同:

1. 大型系统经常是由独立的、交互的子系统组成。不同的团队独立开发不同的子系统。通常, 这些团队在不同的地点甚至不同的时区工作。让每一个团队都去了解整个系统是不太现实的。所以, 他们的重点通常是完成他们自己的那部分, 而不用太关心系统的其他部分。
2. 大型系统是“brownfield systems” (Hopkins and Jenkins, 2008); 也就是说他们包含了一系列的已存在的系统并与它们进行交互。许多系统需求关注这种交互, 因此不是真的适应于灵活性和增量式开发。政治问题也是很重要的, 通常一个问题的最简单解决方案就是变更现有的系统。但是, 这需要与系统的管理人员谈判, 使他们相信这些变更对系统操作没有风险。
3. 当一个系统是由多个系统集成产生时, 开发工作中一个重要部分就是系统配置而不是原始代码的开发。这不一定与增量式开发和频繁的系统集成兼容。
4. 大型系统和他们的开发流程通常受限于外部规则和规章限制, 比如要求写各种系统文档等。
5. 大型系统有很长的采购和开发时间。很难保持团队在整个周期中保持对系统有连贯的认识, 不可避免的会有人转到其他工作或项目中。
6. 大型系统通常具有不同的信息持有者。比如, 护士和管理员可能是这个医疗系统的终端

用户，但是高级医护人员、医院经理等也是信息持有者。将所有这些不同的信息持有者加入到开发流程中当然很重要。

伸缩的敏捷方法有两个观点：

1. 照搬放大的观点，关注如何将这些方法应用到那些小团队无法开发的大型项目中。
2. 渗透的观点，关注如何将敏捷方法介绍推广到拥有多年开发经验的大机构中。

敏捷方法必须去适应大型软件工程。Leffingwell (2007) 认为保留敏捷方法的基本内涵是非常重要的——弹性计划、频繁发布、持续集成、测试驱动开发、良好的团队沟通。必须引入如下一些关键性的调整：

1. 对大型系统，不可能只关注系统的代码。你需要做更多预先的设计和系统文档。软件体系结构必须被设计好，必须要用文档描述系统的一些关键面，比如数据库模式，不同团队的工作划分等。
2. 跨团队沟通机制必须建立和使用。这包括团队成员间常规的电话和视频会议，团队间用于更新各自进度的频繁短暂的电子会议。包括 E-mail、即时消息、wiki 和社交网络等沟通渠道可以为沟通提供便利。
3. 持续集成，当系统是由很多个程序模块组成时，由于任何一个时刻都会有程序员正在处理一个变更，所以，持续的集成在实际过程中是不可行的。但是，保持频繁的系统构建和定期地发布系统是十分关键的。这可能意味着需要引入能支持多团队软件开发的新的配置管理工具。

小软件公司非常热衷于采用敏捷方法。这些公司不受机构繁文缛节或过程标准的约束，他们可以快速采纳新思想。当然，大型公司也在一些特定项目中尝试敏捷方法，但是将敏捷推广到整个公司对他们来说比较困难。Lindvall 等 (2004) 讨论了在 4 个大型技术公司中推广伸缩式敏捷方法时遇到的问题。

在大型公司中很难引入敏捷方法的原因有以下几方面：

1. 没有敏捷开发经验的项目经理可能不愿意接受新方法带来的风险，因为他们不知道这会给他们的项目带来什么样的影响。
2. 大型公司通常有质量保证流程和标准，每一个项目都需要去遵循，他们的官僚性质，很可能跟敏捷是不兼容的。有时候，这些是有软件工具支持的（比如需求管理工具），而这些工具的使用又是必须的。
3. 当团队成员技术都比较高时，敏捷方法似乎能工作得更好。然而，在大型机构中，员工的技术和能力很可能是参差不齐的。那些技术水平比较低的员工在敏捷过程中可能成为无效的员工。
4. 可能从文化上抵制敏捷开发，特别是那些长久使用传统软件工程流程的机构。

变更管理流程和测试流程可能是公司中跟敏捷不兼容的流程的例子。变更管理是控制系统变更的流程，以便变更的影响是可预知的，代价是可控的。所有的变更在做出之前都必须先被审核，这跟重构的观念是冲突的。在极限编程方法中，任何开发人员都可以改进任何代码而不需要其他人同意。对于大型系统，测试是在系统构建后交给外部测试团队完成的。这可能与 XP 中使用的测试优先和经常测试的方法冲突。

引进和维持敏捷在大型机构中的使用是一个文化改变的过程。文化改变需要一段很长的时间来实现，而且通常需要先改变管理方式才能实现。大型机构中希望应用敏捷，需要传道者去推动改变。他们必须投入很多资源去改变流程。当作者在写作本文的时候，很少有大型公司已经实现了在整个机构中向敏捷开发的成功转型。

要点

- 敏捷方法是一种专注于快速开发的增量式开发，频繁地发布软件、降低过程开销、生产高质量的代码。他们使用户直接地参与到开发过程中。
- 决断是否是使用敏捷或计划驱动的方法取决于所开发系统的类型、开发团队的能力和开发系统的公司的文化。
- 极限编程是一种著名的敏捷方法，它集成了一系列好的编程经验，例如频繁地软件发布，连续软件改善和客户参与到软件开发团队。
- 极限编程的一个特别长处是在创建程序特征之前开发自动测试。在增量集成进系统的时候所有的测试必须成功执行。
- Scrum 方法是一种提供项目管理框架的敏捷方法。它的核心是一组冲刺循环，开发一个系统增量是有固定的时间周期的。规划是基于积压的工作的优先权安排的，选择高优先权的任务开始冲刺循环。
- 伸缩的敏捷方法对于大型系统是很困难的。大型系统需要前期设计和一些文档。在实践中，当一个项目有几个独立的开发团队时，连续的集成是不可能的。

进一步阅读材料

《Extreme Programming Explained》这是第一本有关 XP 的书，而且也许至今仍然是最可读的书了。本书从 XP 的发明者的角度解释了此方法，从中能清楚地流露出他的热情（Kent Beck, Addison - Wesley, 2000）。

《Get ready for agile methods, with care》对敏捷方法的一个颇有见地的评论，讨论了它们的长处和短处，是由具有大量经验的软件工程师所写（B. Boehm, IEEE Computer, January 2002）。<http://doi.ieeecomputersociety.org/10.1109/2.976920>。

《Scaling Software Agility: Best Practices for Large Enterprises》这本书主要介绍了伸缩的敏捷开发，也介绍了一些基本的敏捷方法，如 XP, Scrum 和 Crystal（D. Ćeffingwell, Addison-Wesley, 2007）。

《Running an Agile Software Development Project》许多关于敏捷方法的书致力于一种特殊的方法，但这本书采用了一个不同的方法，并讨论了怎样在项目的实际中应用 XP 方法（M. Holcombe, John Wiley and Sons, 2008）。

练习

- 3.1 解释为什么对新系统来说，快速交付和部署要比这些系统的具体功能更重要。
- 3.2 解释敏捷方法的基本原理是如何能带来加速的软件开发和部署。
- 3.3 你在什么情况下会劝告不用敏捷方法来开发软件系统？
- 3.4 极限编程是用故事情节来表达用户需求的，每一个情节书写在卡片上。讨论这种需求描述方法的优点和缺点。
- 3.5 解释为什么测试优先的开发能帮助程序员获得对系统需求的更好的理解。测试优先开发有什么潜在的困难？
- 3.6 给出 4 个理由说明为什么结对编程的软件生产率比程序员单个编程时高。
- 3.7 比较 Scrum 和第 23 章介绍的常规的计划驱动的项目管理方法。比较应基于每种方法在以下几方面的有效性：计划人员的分配，估计项目的预算，维持团队的凝聚力，管理项目团队成员的变更。

- 3.8 假设你是公司里的软件管理者，为飞机开发了一个安全性要求极高的控制软件。你负责开发软件设计支持系统的项目，此系统是要支持将软件需求翻译成形式化软件描述（第13章介绍的）。请评论下列开发策略的优点和缺点：
- 从软件工程师和其他的信息持有者（如管理认证机关）处收集需求，并用计划驱动的方法开发系统。
 - 用一种脚本语言开发原型，如 Ruby 或 Python，与软件工程师和其他的信息持有者评估该原型，然后审查系统需求。使用 Java 开发最终的系统。
 - 使用敏捷方法并让用户参与到开发团队中开发此系统。
- 3.9 让一名用户紧密参与到软件开发团队中的一个问题是“地方化”，也就是，团队成员采纳开发团队的观点而忽视用户队员的需求。请写出3个解决建议，并讨论每一个的优点和缺点。
- 3.10 为了降低成本和交通环境的影响，你的公司决定关闭一些办公室，并为员工提供在家工作的支持。但是，引入该策略的高级管理人员不知道该软件是依赖团队密切合作和结对编程的敏捷方法的。讨论这项新策略可能导致的困难，并写出你将如何应对这些问题。

参考书目

- Ambler, S. W. and Jeffries, R. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons.
- Arisholm, E., Gallis, H., Dyba, T. and Sjöberg, D. I. K. (2007). 'Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise'. *IEEE Trans. on Software Eng.*, **33** (2), 65–86.
- Astels, D. (2003). *Test Driven Development: A Practical Guide*. Upper Saddle River, NJ: Prentice Hall.
- Beck, K. (1999). 'Embracing Change with Extreme Programming'. *IEEE Computer*, **32** (10), 70–8.
- Beck, K. (2000). *extreme Programming explained*. Reading, Mass.: Addison-Wesley.
- Carlson, D. (2005). *Eclipse Distilled*. Boston: Addison-Wesley.
- Cockburn, A. (2001). *Agile Software Development*. Reading, Mass.: Addison-Wesley.
- Cockburn, A. (2004). *Crystal Clear: A Human-Powered Methodology for Small Teams*. Boston: Addison-Wesley.
- Cockburn, A. and Williams, L. (2001). 'The costs and benefits of pair programming'. In *Extreme programming examined*. (ed.). Boston: Addison-Wesley.
- Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum*. Boston: Addison-Wesley.
- Demarco, T. and Boehm, B. (2002). 'The Agile Methods Fray'. *IEEE Computer*, **35** (6), 90–2.
- Denning, P. J., Gunderson, C. and Hayes-Roth, R. (2008). 'Evolutionary System Development'. *Comm. ACM*, **51** (12), 29–31.
- Drobna, J., Nofzt, D. and Raghu, R. (2004). 'Piloting XP on Four Mission-Critical Projects'. *IEEE Software*, **21** (6), 70–5.
- Highsmith, J. A. (2000). *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House.
- Hopkins, R. and Jenkins, K. (2008). *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston, Mass.: IBM Press.

- Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Englewood Cliff, NJ: Prentice Hall.
- Leffingwell, D. (2007). *Scaling Software Agility: Best Practices for Large Enterprises*. Boston: Addison-Wesley.
- Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., May, J. and Kahkonen, T. (2004). 'Agile Software Development in Large Organizations'. *IEEE Computer*, **37** (12), 26–34.
- Martin, J. (1981). *Application Development Without Programmers*. Englewood Cliffs, NJ: Prentice-Hall.
- Massol, V. and Husted, T. (2003). *JUnit in Action*. Greenwich, Conn.: Manning Publications Co.
- Mills, H. D., O'Neill, D., Linger, R. C., Dyer, M. and Quinnan, R. E. (1980). 'The Management of Software Engineering'. *IBM Systems J.*, **19** (4), 414–77.
- Moore, E. and Spens, J. (2008). 'Scaling Agile: Finding your Agile Tribe'. *Proc. Agile 2008 Conference*, Toronto: IEEE Computer Society. 121–124.
- Palmer, S. R. and Felsing, J. M. (2002). *A Practical Guide to Feature-Driven Development*. Englewood Cliffs, NJ: Prentice Hall.
- Parrish, A., Smith, R., Hale, D. and Hale, J. (2004). 'A Field Study of Developer Pairs: Productivity Impacts and Implications'. *IEEE Software*, **21** (5), 76–9.
- Poole, C. and Huisman, J. W. (2001). 'Using Extreme Programming in a Maintenance Environment'. *IEEE Software*, **18** (6), 42–50.
- Rising, L. and Janoff, N. S. (2000). 'The Scrum Software Development Process for Small Teams'. *IEEE Software*, **17** (4), 26–32.
- Schwaber, K. (2004). *Agile Project Management with Scrum*. Seattle: Microsoft Press.
- Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice Hall.
- Smits, H. and Pshigoda, G. (2007). 'Implementing Scrum in a Distributed Software Development Organization'. *Agile 2007*, Washington, DC: IEEE Computer Society.
- Stapleton, J. (1997). *DSDM Dynamic Systems Development Method*. Harlow, UK: Addison-Wesley.
- Stapleton, J. (2003). *DSDM: Business Focused Development, 2nd ed.* Harlow, UK: Pearson Education.
- Stephens, M. and Rosenberg, D. (2003). *Extreme Programming Refactored*. Berkley, Calif.: Apress.
- Sutherland, J., Viktorov, A., Blount, J. and Puntikov, N. (2007). 'Distributed Scrum: Agile Project Management with Outsourced Development Teams'. 40th Hawaii Int. Conf. on System Sciences, Hawaii: IEEE Computer Society.
- Weinberg, G. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand.
- Williams, L., Kessler, R. R., Cunningham, W. and Jeffries, R. (2000). 'Strengthening the Case for Pair Programming'. *IEEE Software*, **17** (4), 19–25.

需求工程

目标

本章的目标是要介绍软件需求的概念，并讨论与发现和记录这些需求所需的过程。当你读完本章，你将了解以下内容：

- 了解用户需求和系统需求的概念以及这些需求要使用不同的方法表达的原因；
- 了解功能需求和非功能需求之间的不同；
- 了解如何在软件需求文档中机构需求；
- 了解导出、分析和有效性验证等主要的需求工程活动，以及这些活动之间的关系；
- 了解需求管理为什么是必要的，以及它是如何支持其他需求工程活动的。

对系统应提供的服务和所受到的约束的描述就是系统需求的内容。这些需求反映了客户对系统帮助其解决某些问题的需要，这些问题可以是：对设备的控制，安排一次订货，或者是信息的查询，等等。对服务和约束的发现、分析、建立文档、检验的过程叫做需求工程。

需求这个术语在软件行业中使用得可能很不一致。在某些情况下，一个需求被视为对系统应该提供的服务或对系统的约束的一个高层抽象描述，而在另一些较极端的情形，它又被定义为是对系统功能的详细的、用数学方法的形式化描述。Davis（1993）解释了为什么会存在这些不同。

如果一家公司要与另一机构签订一个大型软件开发项目合同，那这家公司就要尽量概要地定义对该项目的要求，而且在这样的描述中不应限制解决方案。这时候，就需要一个文本形式的需求以便多个承包商竞标，或者以不同方式满足客户的机构需求。一旦签订了合同，承包商就要为客户写出更详细的系统定义，要让用户能看懂并要在此确认系统到底需要提供哪些服务。这两种文件都被称为需求文档。

某些出现在需求工程过程期间的问题就是因为没有对这两个层次的描述做出清晰的分离。本书采用用户需求这个术语来表达高层的概要需求，用系统需求这个术语表达对系统应该提供哪些服务的详细描述。用户需求、系统需求的定义如下：

1. 用户需求是用自然语言加图的形式给出的、关于系统需要提供哪些服务以及系统操作受到哪些约束的声明。

2. 系统需求详细地给出系统将要提供的服务以及系统所受到的约束。系统需求文档有时也称为功能描述，应该是精确的。它可能成为系统买方和软件开发者之间合同的重要内容。

因为需要向不同类型的读者传达系统信息，因此，不同层次的需求是有用的。图 4-1 说明了用户和系统需求之间的区别。这个来自心理健康护理病人管理系统（MHC-PMS）的实例给出了怎样将一个用户需求扩展为一系列系统需求。从图 4-1 我们可以看出，用户需求是十分概括的，而系统需求对即将要实现的系统所应该提供的服务和功能给出了更加详尽的信息。

我们需要写不同详细程度的需求，因为不同的读者以不同的方式来阅读这些需求。图 4-2 给出了关于用户需求和系统需求的潜在读者。用户需求的读者一般不关心系统是如何实现的，他们很可能就是管理层的人，对系统的技术细节不是很感兴趣。而系统需求的读者需要了解关于系

用户需求定义
1. MHC-PMS 系统每月都要产生报告来显示当月每个诊所所开药物的成本。
系统需求描述
1.1 在每个月的最后一个工作日，对所开的药物、药物成本以及开药诊所应该产生一个汇总报告；
1.2 系统应该在当月最后一个工作日的 17：30 之后自动生成打印报告；
1.3 应该为每一个诊所生成一个报告，并列出各种药的药名、处方的总数、药物总量，以及所开药物的成本；
1.4 如果药物可以分为不同剂量单位（如 10 mg，20 mg），就要为每一剂量单位单独生成报告；
1.5 对所有成本报告的访问权限应只限于管理访问控制单上的授权用户。

图 4-1 用户需求和系统需求

统如何工作的更详细的内容，因为他们关心系统如何将业务过程，或者是因为他们参与了系统的具体实现。

本章介绍的是一种传统意义上的需求，而不是在敏捷过程中的需求。对于大多数大型系统来说，仍然属于这种情况，就是在系统实现开始之前有一个清晰明确的需求工程阶段。所产生的结果是一个需求文档，它可能是系统开发合同的一部分。当然，通常以后会有对需求的变更，用户需求可能会扩展为更多具体的系统需求。然而在系统开发过程中同时导出需求的敏捷方法对于大型系统开发来说是很少用到的。

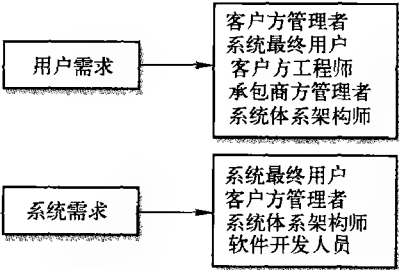


图 4-2 不同类型的需求描述的读者

4.1 功能需求和非功能需求

软件系统需求常常分为功能需求和非功能需求：

- 1. 功能需求 包括对系统应该提供的服务、如何对特殊输入做出反应，以及系统在特定条件下的行为的描述。在某些情况下，功能需求可能还需明确声明系统不应该做什么。
- 2. 非功能需求 对系统提供的服务或功能给出的约束。包括时间约束、开发过程的约束和所受到的标准的约束。非功能需求经常适用于整个系统而不是个别的系统功能或服务。

事实上，这些不同类型的需求之间的区别并不像定义的那么明显。若用户需求是关于信息安全性的，例如对授权用户的访问限制的声明，则表现为一个非功能需求。然而，当具体开发时，它可能导致其他明显的功能性的需求，比如，包括系统中用户授权的需求。

这些表明需求并不是独立的，一个需求经常会产生或是约束其他的需求。因此，系统需求并不仅仅是要具体说明系统所需要提供的服务或功能，也必须明确指明确保这些服务和功能正确交付的一些必要的功能。

4.1.1 功能需求

功能需求描述系统所提供的功能或服务。它取决于开发的软件类型、软件潜在的用户，以及机构在写需求时所采用的一般方法。如果是用户需求，就要用可以被系统用户理解的一种抽象方法来描述功能需求。然而更具体的功能性系统需求则需要详细地描述系统功能、输入和输出、异常等。

功能性系统需求是多方面的，从系统所应提供的服务这样的一般性需求到反映本地的特色工作方式或是机构已有系统这样一些特殊需求。下面是 MHC-PMS 系统的几种功能需求，此系统是用来维护进行心理健康护理的病人信息的：

1. 用户能够搜索到所有诊所的预约挂号单。
2. 系统每天能够为每个诊所生成一份想预约在那天就诊的病人名单。
3. 每位使用该系统的职员都可以通过他的8位雇员号码被唯一地识别。

这些功能性用户需求定义了系统必须提供的特殊功能。这些需求是从系统的用户需求文档中摘取出来的,从中可以看出功能需求可能会以不同的详细程度被重写(对比一下需求1和需求3)。

软件工程中的许多问题都源自对需求描述的不严密。系统开发者自然想把模糊的需求以某种容易实现的内容去解释它。然而,客户常常不希望这样做。新的需求需要不断地建立,需求变更也会不断地发生。当然,这样会延迟系统的交付,也会增加成本。



领域需求

系统的领域需求是从系统的应用域中得出的而不是从个别系统用户那里得到的。它们可能本身就是新的功能需求,也可能是约束现有的功能需求,或者是特别的计算必须如何进行的规定。

领域需求的问题在于,软件工程师可能对系统所运行的域的特点没有了解。他们无法掌握是否领域需求被遗漏或是与其他的需求发生冲突。

<http://www.SoftwareEngineering-9.com/Web/Requirements/DomainReq.html>

例如,MHC-PMS系统的第一个需求例子中提到用户能够找到所有诊所的预约挂号单。有这种需求的理由是有心理健康问题的病人经常是糊涂的。他们可能预定了一家诊所而实际上却去了另外一家。如果他们有一个预约,他们将会被记录为已就诊,而无论是在哪个诊所。

医务工作人员定义“搜索”的含义是希望通过给出病人的姓名,系统就可以在所有诊所的预约单中查找该姓名。尽管如此,这在需求中并不是阐述得很直白。系统开发者可能会用另一种不同的方式来解释这个需求,而且会实现一个搜索功能以至于用户不得不选择一家诊所才可以执行搜索功能。这明显会涉及更多的用户输入而花费更多的时间。

理论上,系统的功能需求描述应该既完备又一致。完备意味着用户所需的所有服务都应该给出描述。一致意味着需求描述不能前后矛盾。在实际中,对大型而又复杂的系统而言,要做到需求描述既完备又一致几乎是不可能的。一方面是因为在为复杂系统写需求描述时很容易出现错误和遗漏;另一方面是因为在一个大型系统中有很多信息持有者,一个信息持有者是在一定程度上被系统影响的一个人或一个角色。这些信息持有者经常有着不同和不一致的需求。在最初描述需求的时候,这些矛盾可能不明显,需求的不一致性就这样潜伏在描述中了。只有深入地分析之后或当系统交付客户使用之后问题才能暴露出来。

4.1.2 非功能需求

所谓非功能需求,如名字所示,是指那些不直接关系到系统向用户提供的具体服务的一类需求。它们与系统的总体特性相关,如可靠性、响应时间和储存空间占用等。换言之,它们对系统的实现定义了约束,如I/O设备的能力、与其他系统接口的数据的表示等。

非功能性系统需求,例如性能,安全性,可用性,通常会从总体上规范或约束系统的特性。非功能性需求通常会比个别的功能性需求更加关键。系统用户经常发现所用系统并没有在某个功能方面满足他们的需求,但是还是能想办法克服这些不足。然而,如果一个非功能系统需求没有满足则可能使整个系统无法使用。举例来说,如果一个飞机系统不符合可靠性需求,它将不会被批准飞行;如果一个实时控制系统无法满足其性能需求,控制功能可能根

本无法使用。

识别哪个系统组件实现特别的功能性需求是比较容易的（例如那些实现报告需求的格式化组件），但是把这些组件与非功能性需求联系起来却是相当困难的。这些需求的实现可能散布在整个系统之中。原因如下：

1. 非功能性需求会影响整个系统的体系结构，而不是个别的组件。例如，为了保证系统的性能需求，就必须合理组织系统使得组件之间的通信量达到最小。

2. 单个的非功能需求，比如一个信息安全性需求，可能会产生数个相关的功能性需求，这些功能性需求定义了新系统所要求的服务。

非功能需求源于用户的需要，因为预算约束、机构政策、与其他软硬件系统的互操作，还包括如安全规章、隐私权保护的法律效力等外部因素。图 4-3 是对非功能需求的一个分类。从这个分类图中可以看出非功能需求或是来源于所要求的软件特性（产品需求），或是来源于开发软件的机构（机构需求），或是来源于外部来源。

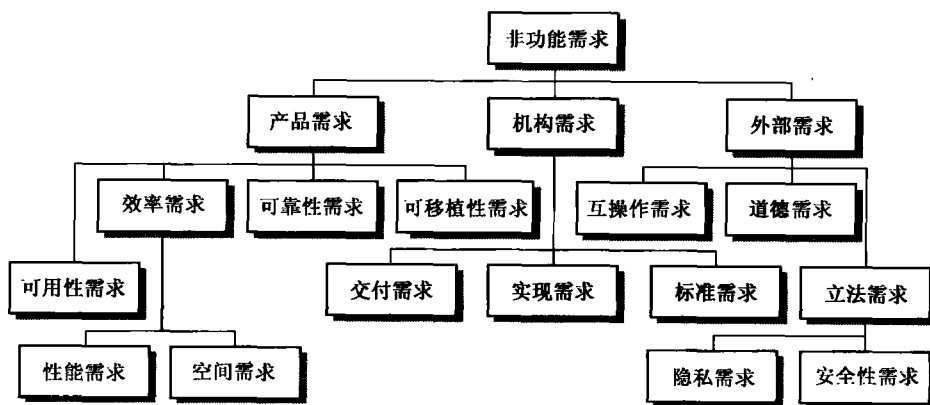


图 4-3 非功能需求的类型

1. **产品需求** 这些需求定义或约束软件的行为。例子包括系统执行速度有多快和内存消耗需要多少等性能需求，包括指定系统可以接受的出错率等系统可靠性需求，也包括信息安全性需求和可用性需求。

2. **机构需求** 这些需求是很广泛的系统需求，起源于客户所在的机构和开发者所在的机构中的政策和规定。例子包括过程标准，即机构中采用的过程标准；实现要求，如所采用的程序设计语言和设计方法；交付需求，即有关对产品及其文档交付的要求。

3. **外部需求** 这也是个广泛的类别，包括所有来自于系统外部因素和开发过程的需求。这些需求会包括监管需求，指定什么是系统必须实现的以通过如中央银行这样的上级监管部门的批准；立法需求，必须得到遵守以确保系统在法律许可的范围内工作；道德需求，保证系统能被用户和一般社会公众所接受。

图 4-4 给出了来自 MHC-PMS 系统的产品需求、机构需求和外部需求的实例，它的用户需求请参见 4.1.1 节。它的产品需求是一种可用性需求，定义了系统每天的可用性和不可用性时间。没有提及 MHC-PMS 的功能，但却明确地规定了系统设计人员必须要考虑的约束。

机构需求规定了系统如何验证用户。对于所用的软件来说，操作系统的权力正在移向一个标准化的验证程序，不再是用户拥有一个登录名，而是通过一个阅读器来刷身份证以此来验证身份。外部需求来自于对隐私权保护的法律条文的遵守，隐私在医疗系统中显然是一个非常重要的问题。外部需求规定系统必须与国家隐私标准保持一致。

产品需求
MHC-PMS 系统必须对所有诊所在正常工作时间内（周一至周五，08：30 ~17：30）都是可用的。任何一天的正常工作时间内系统的关闭时间不应超过5秒钟。
机构需求
MHC-PMS 系统的用户应该使用他们的卫生局身份证件来验证自己。
外部需求
系统必须依照法律 HStan - 03 - 2006 - priv 贯彻病人隐私条款。

图 4-4 MHC-PMS 中的非功能需求实例

一个存在于非功能性需求中的普遍问题是，用户或消费者经常建议把这些需求作为总的目标，比如易用性，系统的恢复性，或是快速的反应能力。目标固然能提出好的计划，但也带给开发者许多问题，因为这会给系统交付之后在客户和开发者之间引发争议。例如，下面的系统目标就是一个管理者如何表达可用性需求的典型例子。

系统对医务工作人员来说应该是容易使用的，并且以一种用户错误最小的方式来管理系统。

再次提到这个需求是为了说明该目标怎样才能表达为一种“可测试”非功能需求。客观地来验证系统目标是不可能的，但是在以下的描述中你至少可以用软件手段计数由用户在测试系统时所制造出的错误。

医务工作人员应该能够在4小时的培训后学会使用系统的所有功能。在培训后有经验的用户在系统的使用中所犯的错误不能超过2个/小时。

只要有可能，我们就应该使非功能需求得以量化，从而使其验证更客观。图4-5给出了许多可能用来指定非功能性系统属性的量度。基于这些度量的测试可以检验系统是否满足了相应的需求。

属 性	度 量
速度	处理的交易/秒 用户/事件响应时间 屏幕刷新时间
规模	兆字节数 ROM 芯片数
易用性	培训时间 帮助帧数
可靠性	平均失败时间 不可用的概率 失败发生频率 可用性
鲁棒性	失败后重启时间 事件引起失败的百分数 失败中数据崩溃的概率
可移植性	目标依赖语句的百分数 目标系统的数目

图 4-5 定义非功能需求的量度

在实际过程中，对需求描述的量化通常是很困难的。客户没有能力将目标转化成量化的需求，像可维护性这样的目标，没有量度可供使用。在另外一些情形中，即使量化描述是可能做到的，客户也没有能力把他们的需求和这些描述相对应。他们根本不理解一个描述所需要的指标，比如说可靠性，和他们日常计算机操作经验有什么联系。对非功能需求的量化验证成本极高，支

付系统开发的客户会认为这些成本是不划算的。

非功能需求常与功能需求或其他功能需求发生冲突，它们之间存在着相互作用关系。举例来说，在图 4-4 的验证需求中显然要求每台电脑要安装一个连接该系统的读卡器。有时候要从医生或护士的笔记本上移动性访问该系统，而它们是不能正常安装读卡器的。在这种情况下，就必须有可选验证方法。

实际上，将功能需求和非功能需求在需求文档中区分开是很困难的，如果将非功能需求从功能需求中分开，它们之间的关系就很难看出来。然而，对系统总体特性方面的需求应该直白地突出显示出来，例如性能或可靠性。即用需求文档中的单独一部分来描述，或者用其他方式表示以区别于其他系统需求。

非功能性需求，如可靠性、安全性和信息安全性需求，对于要求极高的系统来说至关重要。因此第 12 章将对可靠性和安全性需求相关技术做更详尽的描述和讨论。



需求文档标准

有很多大型机构，例如美国国防部和 IEEE，它们对需求文档有定义好的标准。这些标准是非常一般化的，但尽管如此，作为开发更详细的机构自己的标准是很有用的。美国电气电子工程师协会（IEEE）是一个闻名的标准提供者，它们已经开发了需求文档结构。该标准对于像军事指控系统这样具有长生命期并总是由一群开发机构所开发的系统来说是最为合适的了。

<http://www.SoftwareEngineering-9.com/Web/Requirements/IEEE-standard.html>

4.2 软件需求文档

软件需求文档（有时叫做软件需求描述或 SRS）是对系统开发者需要实现什么的正式陈述。它应该包括系统的用户需求和一个详细的系统需求描述。在某些情况下，用户需求和系统需求被集中在一起描述。在其他的情况下，用户需求在系统需求的引言部分给出。如果有很多的需求，详细的系统需求可能被分隔到不同文档中单独描述。

在外部承包商开发软件系统时需求文档是必要的。然而敏捷开发模式的使用表明由于需求的快速变化，致使需求文档在写完时已经过时，也就浪费了大量的精力。于是像极限编程（Beck, 1999）这类的方法相应产生，这种方法是增量式收集用户需求，并把它们作为用户故事情景写在卡片上。然后用户对要实现的需求给出优先级排序，最为紧要的需求将在下一个增量中优先考虑。

作者认为，这种方法很适合需求不稳定的业务系统。但是有一份定义系统的业务和可靠性需求的短的支持文档仍然是有用的。当专注于系统下一个版本的功能性需求时，很容易忘记应用到整个系统上的需求。

需求文档有一个较广范围的读者群，从那些订购系统的高级机构管理者到负责开发系统的软件工程师。图 4-6（Kotonya 和 Sommerville, 1998）说明了文档的可能用户和他们如何使用文档。

可能用户的广泛性意味着需求文档必须在以下几

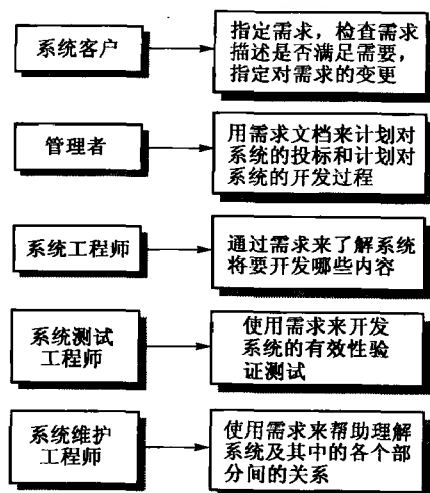


图 4-6 需求文档的用户

个方面采取一个折中：与客户关于需求的沟通；为开发者和测试者在细节层次上定义需求；附带可能对系统所做的进化的有关信息。对可预见变更方面的信息能帮助系统设计者避免做出一些苛刻的设计决策，也能帮助系统维护工程师避免不得不为新需求而去调整系统。

需求文档中内容的详细程度，取决于所要开发的系统的类型以及所使用的开发过程。要求极高的系统需要有详尽的需求，因为安全性和信息安全性都需要详细的分析。当一个系统是由某个外部机构承担的时候，要求极高的一类系统的描述就需要非常精确和详细。如果需求中有较大的弹性的话，且系统是由本机构内部开发的话，文档就不必要写得太详细，一些二义性问题可以在开发阶段得以解决。

图 4-7 是一个基于 IEEE 标准的需求文档的结构。IEEE 标准是一个通用的标准，可以调整以适应特殊场合。这里是对 IEEE 标准的扩展，包含了由 Heninger 提出的系统进化预测的有关内容。此内容将有助于系统的维护人员，允许系统设计人员加入对未来系统特点的支持。

章 节	描 述
绪言	定义文档的读者对象，说明版本的修正历史，包括新版本为什么要创建，每个版本间的变更内容的概要
引言	应该描述为什么需要该系统，简要描述系统的功能，解释系统是如何与其他系统协同工作的。要描述该系统在机构总体业务目标和战略目标中的位置和作用
术语	定义文档中的技术术语和词汇。假设文档读者是不具有专业知识和经验的人
用户需求定义	这一部分要描述系统应该提供的服务以及非功能系统需求，该描述可以使用自然语言、图表或者其他各种客户能理解的标记系统。产品和过程必须遵循的标准也要在此定义
系统体系结构	这一部分要对待建系统给出体系结构框架，该体系结构要给出功能在各个模块中的分布。能被复用的结构中组件要用醒目方式示意出来
系统需求描述	这一部分要对功能和非功能需求进行详细描述。如有必要，对非功能需求要再进一步描述，例如，定义与其他系统间的接口
系统模型	这一部分要提出一个或多个系统模型，以表达系统组件、系统以及系统环境之间的关系。这些模型可以是对象模型、数据流模型和语义数据模型
系统进化	这一部分要描述系统基于的基本设想和定位以及硬件和用户需求改变时所要做的改变。这部分对系统设计人员来说是有用的，因为这有助于他们避免一些设计决策，这些决策可能会限制未来系统的变更
附录	这一部分要提供与开发的应用有关的详细、专门的信息。该附录的例子是硬件和数据库的描述，硬件需求定义了系统最小和最优配置，数据库需求定义了系统所用的数据的逻辑结构和数据之间的关系
索引	可以包括文档的几个索引。除了标准的字母顺序索引外，还可以有图表索引、功能索引等

图 4-7 需求文档的结构

当然，一个需求文档中的内容是和被开发软件的类型以及开发中使用的方法紧密相关的。如果一个进化式方法用在一种软件产品开发上，需求文档将会省去上面提到的许多有关细节。重点也将会被放在用户需求的定义和高标准的非功能需求上面。在这种情况下，设计者和编程人员将根据他们的判断来决定如何设计系统以满足用户的需求。

然而，当软件系统是大型系统工程项目的一部分时，大系统本身包含交互式硬件和软件系统，一般就必须在细粒度层次上定义需求。这意味着需求文档会非常长，而且可能包含图 4-7 中所列的大部分章节。对于长文档，尤其需要一个详细的目录和文档索引，以便读者能快速找到所需的信息。



使用自然语言进行需求描述时的问题

自然语言描述需求是很有用的，但是它的灵活性也总带来问题。为描述不清晰的需求提供了空间，阅读者（设计人员）也会错误解释需求，因为他们有不同的背景，对系统用户的认识是很不相同的。很容易将多个需求混合在一个句子中，而对自然语言描述的需求的结构化会有很大的难度。

<http://www.SoftwareEngineering-9.com/Web/Requirements/NL-problems.html>

4.3 需求描述

需求描述就是在需求文档中写下用户需求和系统需求。在理想情况下，用户需求和系统需求应该具有清晰性、明确性、易读性、完整性和一致性。在实际中，这是很难做到的，因为信息持有者会用不同方式解释需求，而且在需求中常有内在的冲突和不一致性。

用户需求是从用户角度来描述系统功能需求和非功能需求，以便让不具备专业技术方面知识的用户能看懂。这样的需求描述只描述系统的外部行为，要尽量避免对系统设计特性的描述。因而，用户需求就不可能使用任何实现模型来描述，而是用自然语言、图形来叙述。

系统需求是用户需求的扩展版，是软件工程师开始系统设计的起点。系统需求添加了许多细节内容，解释如何能让系统提供用户需求。它们可以作为有关系统实现合同的一部分，因此它们是对系统的一个完备且详尽的描述。

原则上讲，系统需求应该仅仅描述系统的外部行为和对它的操作上的限制，而不应该包括系统应该如何设计和如何实现。然而，要在细节层次上给出复杂软件系统的完善的定义，不提到任何设计信息事实上是不可能的。有这样一些理由：

1. 首先要给出系统的初始的体系结构，借助这个框架来梳理需求描述。系统需求依照构成系统的不同子系统结构来给出。像在第6章和第18章中讨论的那样，若是在系统实现中想复用软件组件，体系结构描述就是非常关键的。

2. 在大部分情况下，系统和其他已存在的系统存在互操作。这就约束了系统的设计，同时，这些约束又构成了新系统的需求。

3. 使用特定的架构来满足某些非功能需求（例如，N-版本编程环境有助于提高系统可靠性，将在第13章给出讨论）有时是必要的。会对系统安全进行认证的外部监管者要指定使用一个已经被认证的体系结构。

用户需求大部分是用自然语言来描述，并附加了一些图。系统需求也会使用自然语言来描述，但是会用到其他的一类符号，这些符号基于表格、图形化系统模型，或者数学系统模型。图4-8总结了描述系统需求时可能会用到的一些符号。

在需要描述状态变化或者一系列活动时，图形化模型是非常有用的。在第5章中提到的UML序列图和状态图，显示了某一消息或事件发生时所产生的一系列活动。形式化数学描述有时被用来描述安全要求极高的或是信息安全要求极高的系统，但是在其他情况下很少用到。在第12章中会讲解如何书写需求描述。

4.3.1 自然语言描述

在软件工程初期，自然语言就已经被用来描述软件需求。它易于表达，直观且普遍使用。但同时也具有二义性，文化背景不同的读者所理解的意思也不相同。因此，提出了一些替代的方法

符 号	描 述
自然语言句子	需求是用有标点符号的自然语言句子写的。每个句子应该表达一个需求
结构化的自然语言	需求是用在一个标准格式或模板中的自然语言写的。每个域提供需求的某个方面的信息
设计描述语言	此方法使用一种像程序语言的语言，但是具有更抽象的特征，通过定义一个系统的操作模型来描述需求。该方法现在已经很少使用了，尽管它对接口描述是非常有用的
图形化符号	图形化模型，辅之以文本标记，用于定义系统的功能需求。UML 的用例和序列图是常用的图形表示
数学描述	这些符号是基于数学概念，例如有限状态机或集合。尽管这些无二义的描述能够减少需求文档中的二义性，但是绝大多数的客户没有能力看懂这样的形式描述。他们无法检查需求文档中写的是否代表了他们所希望要的，也就不会情愿接受这样的系统合同形式

图 4-8 系统需求描述的书写方法

来书写需求。尽管如此，这些方法并没有被广泛采用，自然语言将继续作为描述系统和软件需求的最广泛使用的方法。

在使用自然语言书写需求时，为了尽力减少误解，推荐下面一些简单的指导准则：

- 1. 设计一个标准格式，并保证所用的需求定义都遵循此格式书写。标准化格式不易发生遗漏，需求更易检查。使用的格式是用一个单个语句来表达需求。把需求原理和每一个用户需求联系起来解释提出需求的原因。需求原理里面可能也包括关于需求提出者的信息，这样在需要变更需求时就知道该找谁咨询。
- 2. 使用一致性的语言来区分强制性需求和可选性需求。强制性需求是系统必须支持的，定义时要使用“必须”，可选性需求不是必要的，定义时要使用“应该”。
- 3. 对文本加亮（粗体、斜体、颜色）来突出显示关键性需求。
- 4. 不要认为读者会理解技术性软件工程语言。像体系结构、模块之类的语言很容易被误解。因此，要避免使用专业术语和缩写词。
- 5. 在任何可能的情况下，都应该尝试把需求原理和每一个用户需求联系起来。需求原理应该解释需求产生的原因，在需求发生变更时它是尤其有用的，可以用来判断哪些改变是不可取的。

图 4-9 解释了如何使用这些指导方法。它包括两份需求，这两份需求是在第 1 章中介绍的自动化胰岛素泵的嵌入式软件需求。可以从这本书的网站上下载完整的胰岛素泵的需求描述。

3.2 如果需要，系统应该每 10 分钟就测量一下血糖水平和胰岛素的传递量。（血糖的变化是相对缓慢的，因此太过频繁的测量是没有必要的，测量的频率过低又会不必要地导致高血糖。）

3.6 系统应该在被测环境下按表 1 中定义的相关动作每隔一分钟执行一次自测程序。（自测程序能够发现硬件和软件问题，并警告用户正常的操作可能无法实现。）

图 4-9 胰岛素泵软件系统中的需求实例

4.3.2 结构化描述

结构化自然语言是书写系统需求时的一种方法，需求的作者的自由受到限制，所有的需求都要以一种标准方式来书写。这个方法的好处是它保持了自然语言中的绝大部分好的性质，包括表达能力和易懂性，但又对描述施加了一致性的约束。结构化语言使用模板来描述系统需求。描述时可以用程序语言结构中的选择和迭代，也可以使用阴影或不同字体来突出关键部分。

Robertson 和 robertson (Robertson 和 Robertson, 1999) 在他们介绍 VOLERE 需求工程方法一书中, 推荐将用户需求最初写在卡片上, 一个需求单独写在一张卡片上。他们建议在一张卡片上划定多个区域, 如需求原理、对其他需求的依赖关系、需求来源、支持材料, 等等。这类似于在图 4-10 中使用结构化语言描述的例子中所用的方法。

使用结构化方法来描述系统需求, 要先为需求定义一个或多个的标准模板, 并将这些模板表示成结构化的表格形式。描述是结构化的, 一般围绕系统所操作的对象、系统所执行的功能, 或者是系统所处理的事件。图 4-10 是一个基于表格的描述的例子。在这个例子中, 定义了当血糖浓度在安全范围内时计算胰岛素流量的方法。

胰岛素泵/控制软件/SRS/3. 3. 2	
功能	计算胰岛素剂量: 安全的胰岛素水平
描述	计算所要传输的胰岛素剂量, 这是在当前度量的血糖水平处于 3 ~ 7 个单位之间这样正常范围之内时的胰岛素计算
输入	当前血糖读数 (r2), 先前的两个读数 (r0, r1)
来源	来自传感器的当前血糖读数。其他读数来自内存
输出	CompDose - 所要传输的胰岛素剂量
目的地	主控制循环
行动	如果血糖水平是稳定的或是往下掉或是往上升但速率是下降的, 那么 CompDose 是零。如果血糖的水平是在上升且上升的速率也在上升阶段, 那么 CompDose 的计算方法是求当前血糖水平和先前血糖水平, 然后除以 4 并取整。如果取整的结果是零, 那么 CompDose 就被设置成可以传输的最小剂量
需求	两个先前的读数, 这样血糖变化速率就可以计算出来了
前置条件	胰岛素池容纳至少是单个传输剂量的最大值
后置条件	r0 被 r1 替换, 然后 r1 被 r2 替换
副作用	无

图 4-10 胰岛素泵需求的结构化描述

当用一个标准格式描述功能需求时, 下列各项信息应该被包括在内:

- 1. 关于所定义的功能或实体的描述。
- 2. 关于输入及输入来源的描述。
- 3. 关于输出及输出去向的描述。
- 4. 关于计算所需要的信息以及系统中所使用的其他实体的信息 (是“必需”那一部分)。
- 5. 关于所采取的行动的描述。
- 6. 如果使用了一个功能方法, 前置条件设定在此函数被调用之前什么逻辑子句必须为真; 后置条件设定该功能执行之后什么逻辑子句应该为真。
- 7. 关于操作的副作用 (如果有的话) 的描述。

使用结构化的描述去除了自然语言描述中的一些问题, 这是在描述中减少了可变性和需求得到有效组织的结果。然而, 用一种无二义性的描述方法来书写需求有时仍然是很困难的, 尤其是需要描述复杂计算的时候 (例如, 如何计算胰岛素需求)。

为了说明这个问题, 我们可以添加额外信息到自然语言的需求描述中, 方法是用表格或图形模型形式给出描述。通过这些形式, 可以示意出计算是如何进行的, 系统状态是如何改变的, 用户是如何与系统进行交互的, 以及动作是以什么样的顺序执行的。

表格在有多个可能情形时尤其有用, 我们需要对每一个这样的情形描述清楚所要采取的动作

作。胰岛素泵是根据用户的胰岛素需求进行计算的，而用户胰岛素需求是根据自身血糖水平的变化速率得到的。这些变化速率是通过当前和先前一次的读数计算的。图 4-11 描述了如何用血糖的变化速率来计算胰岛素需求量。

条 件	动 作
血糖水平下降 ($r2 < r1$)	CompDose = 0
血糖水平稳定 ($r2 = r1$)	CompDose = 0
血糖水平上升但升速在下降 ($(r2 - r1) < (r1 - r0)$)	CompDose = 0
血糖水平在上升，升速稳定或者也在上升 ($(r2 - r1) \geq (r1 - r0)$)	CompDose = round ($(r2 - r1) / 4$), 如果取整结果为 0 则 CompDose = MinimumDose

图 4-11 胰岛素泵计算的表格描述

4.4 需求工程过程

正如在第 2 章中所提到的，需求工程过程包括有 4 个高层活动。它们是：评估系统是否对业务有用（系统可行性研究），需求发现（需求导出和分析），将需求转变为某种标准格式描述（需求描述），以及检验需求是否正确地定义了客户所希望的系统（需求有效性验证）。在图 2-6 中已经按照过程顺序显示了这些活动。然而，在实际中需求工程是一个活动相互交错的迭代过程。

图 4-12 说明了这种交错性。伴随着系统需求文档的导出，将这些活动组织在一个螺旋结构的迭代过程中。在一次迭代中每个活动所需要的时间和人力依赖于总过程所处的阶段以及所要开发系统的类型。在过程初期，绝大多数人力都是用在高层业务的理解、对非功能需求的理解以及对系统用户需求的理解。在过程的后期，在螺旋的外围，更多的人力将投入到需求导出和理解具体系统需求中。

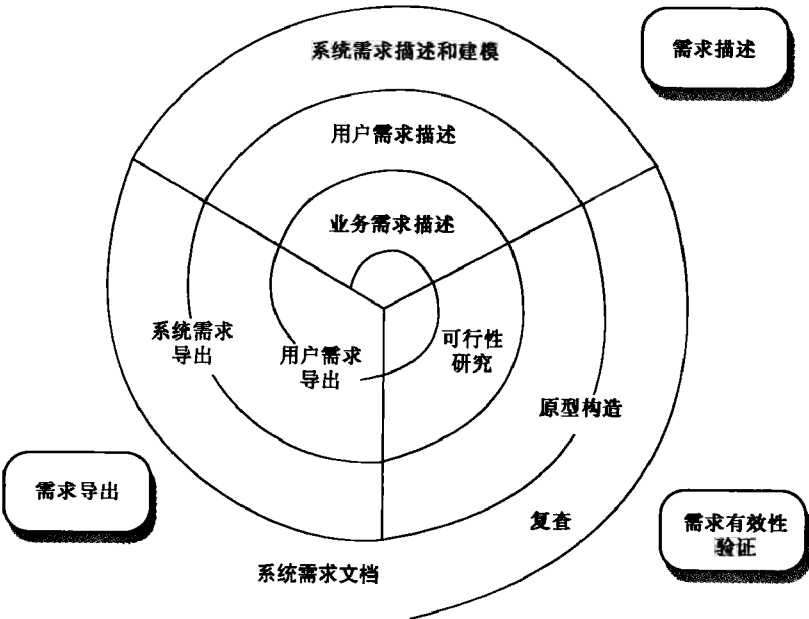


图 4-12 需求工程过程的螺旋模型



可行性分析

可行性分析是一个短的很专注的分析，它应该在需求工程过程的早期阶段进行。应该回答3个关键性问题：a) 系统是否对机构的总体目标有贡献？b) 采用当前技术，系统是否能够在时间要求和预算范围内实现？c) 系统是否能和正在使用中的其他系统集成？

如果对任何一个问题的回答是否定的话，你就应该终止正在进行的项目了。

<http://www.SoftwareEngineering-9.com/Web/Requirements/FeasibilityStudy.html>

螺旋模型所提供的开发方法准许我们将需求处理为不同详细程度。迭代的次数可以不同，这样螺旋就可以在经过若干或所有的用户需求导出后退出。敏捷开发模式可以代替原型开发模式，使得需求和系统实现一同进行。

有些人认为需求工程的任务是要提出一个结构化方法，例如，面向对象的分析（Larman, 2002）。这包括分析系统和开发一组基于图形的系统模型，作为系统的描述，如用例模型。这组模型描述了系统的行为，并用附加的注释信息加以标注解释，如对它的性能或可靠性的要求。

尽管结构化方法在需求工程过程中起到重要的作用，但需求工程的问题远不止于此。尤其是需求导出，它是一个以人为中心的活动，而人是不情愿受到严格的系统模型束缚的。

事实上，所有的系统和需求都是在不断变化的。开发人员都在寻求能够更好地理解软件需求的方法；客户的机构也在变化；系统的硬件、软件和机构环境也在不断改变。管理这些不断变更的需求的过程称为需求管理，4.7节将介绍这一主题。

4.5 需求导出和分析

在初始的可行性研究之后，下一个需求工程过程就是需求导出和分析。在这个活动中，软件开发技术人员要和客户及系统最终用户一起调查应用领域，即系统应该提供什么服务，系统应该具有什么样的性能以及硬件约束等。

需求导出和分析活动可能涉及机构中方方面面的人。系统信息持有者是指所有对系统需求有直接或间接影响力的人。包括将与系统交互的最终用户和机构中其他与系统有关的人员。其他的系统相关人员可能包括正在开发或维护其他相关系统的工程人员、业务主管、领域专家、工会代表等。

需求导出和分析过程的过程模型如图4-13所示。每个机构都会根据这个通用模型产生自己的版本，这依赖于内部因素，包括开发队伍的专业化水平、所开发系统的类型、采用的标准等。

过程活动包括：

1. 需求发现 这是一个与信息持有者交流从而发现他们的需求的过程。来自信息持有者的领域需求和文档也是在这个活动中得以发现的。在这一部分会讲到用于需求发现的一些补充技术。

2. 需求分类和组织 该过程将无序的需求收集起来，对其重新组织和整理，将其分成相关的几个组。对需求归类有一个最常用方法，就是利用系统体系结构模型来识别子系统，并把需求与每一个子系统相关联。事实上，需求工程和体系结构设计不能是完全分离的活动。

3. 需求优先级排序和协商 在有多项目相关人员

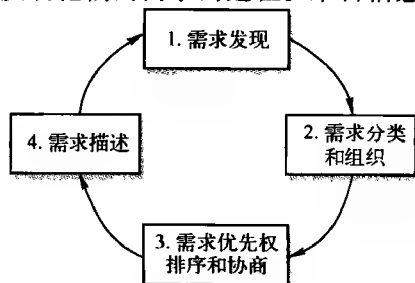


图 4-13 需求导出和分析过程

(信息持有者)参与的地方,需求将不可避免会发生冲突。这个活动就是对需求优先权排序并通过协商发现且解决这些冲突。

4. **需求描述** 记录需求并将它作为螺旋下一循环的输入,产生形式化的或非形式化的需求文档,正如4.3节介绍的那样。

从图4-13可以看出需求导出和分析是一个迭代过程,从一个活动到另一个活动会有持续不断的反馈。过程循环从需求发现开始,以需求文档编制结束。分析人员在每个循环中都能进一步加深对需求的理解。在需求文档完成时循环方可结束。

系统需求导出和分析的过程是一个困难的过程,有这样一些原因:

1. 项目信息持有者除了能通过泛泛的表述外,通常并不真正知道他们希望计算机系统做什么,让他们清晰地表达出他们需要系统做什么是一件困难的事情,因为他们不清楚什么是可行的什么是不可实现的。他们或许会提出不切实际的需求。

2. 系统的信息持有者自然是用他们自己的语言表达需求,这些语言会包含很多他们所从事的工作中的专业术语和专业知识。需求工程师没有客户的领域中的经验,可能并不了解这些需求。

3. 不同的信息持有者有不同的需求,他们可能以不同的方式表达这些需求。需求工程师必须发现所有潜在的需求资源,而且能发现这些需求的相容之处和冲突之处。

4. 政治上的因素可能影响系统的需求。管理者可能提出特别的需求,因为这些将会使他们在机构中增加影响力。

5. 进行需求分析的经济和业务环境是动态的。在分析过程期间它不可避免会变化。因此,个别需求的重要程度可能改变。新的需求可能从新的信息持有者那里得到。

不可避免的是,不同信息持有者对于需求的重要性和优先级的看法是不同的,有时这种不一致意见还可能是冲突的。在这个过程中,你就需要组织信息持有者进行定期协商来达成一致意见。不太可能让每一个信息持有者都十分满意,但是如果某些信息持有者感觉到他们的意见没有得到足够的重视,他们就有可能蓄意去破坏需求工程过程。

在需求描述阶段,所导出的需求以一种有助于进一步需求发现的方式进行记录。在这个阶段,产生系统需求文档的一个早期版本,文档中可能遗漏了某个部分,也可能是不完善的需求。另外一种做法是,用另一种完全不同的方法来记录需求(例如,将需求记录在表格或卡片上)。将需求写在卡片上是十分有效的,因为它们很容易供信息持有者查阅、修改和组织。



视点

视点是一种收集和组织需求的方式,这些需求来自一组具有共同认知的信息持有者。因而每个视点包含一组系统需求。视点可能来自于最终用户、管理者等。视点帮助识别不同的人,这些人能够提供他们的需求的信息。视点也能帮助我们很好地组织需求以便进行分析。

<http://www.SoftwareEngineering-9.com/Web/Requirements/Viewpoints.html>

4.5.1 需求发现

需求发现(有时称为需求导出)是一个对准备建立的系统和正在使用的系统进行信息收集并从这些信息当中提取用户需求和系统需求的过程。需求发现阶段的信息源包括:已有文件,系统信息持有者,以及类似系统的相关描述。我们与信息持有者通过交谈和观察来进行交互,我们也可以使用用例和原型的方法来帮助信息持有者有效地理解系统。

信息持有者范围很广，包括系统的最终用户到管理者，还包括一些外部的信息持有者，比如认证系统可接受性的认证人员。例如，对于心理健康护理病人信息系统来说，系统的信息持有者包括：

1. 病人，他们的个人信息被记录在系统中；
2. 医生，负责评估和治疗病人；
3. 护士，协调医生的会诊和管理一些治疗；
4. 分诊医生，他们管理病人的预约；
5. IT 人员，负责安装和维护系统；
6. 医德管理者，他们必须确保系统满足当前医疗的道德方针；
7. 保健管理者，他们从系统中获取管理信息；
8. 医疗记录保管员，他们要确保系统信息的可维护性、可保留性和执行步骤记录的正确执行。

除了系统信息持有者之外，我们也看到需求还来自于应用领域和其他与本系统有交互关系的系统。所有这些在需求导出过程中都需要考虑到。

这些需求源（信息持有者、领域、系统）都可以用系统视点来表示。这里每一个视点代表一个系统需求的子集。一个问题的不同视点会用不同方法考虑问题，但是这些视点也不是完全独立的，它们经常有所重叠，从而有共同的需求。我们可用这些视点来构建系统的需求发现和文档编制。

4.5.2 采访

对信息持有者的正式的和非正式采访是绝大多数需求工程过程的组成部分。在这些采访中，需求工程的团队成员会向信息持有者提出一系列关于他们所正在使用的系统和将要开发的系统的问题，从他们对这些问题的回答中就能了解系统的需求。采访可能有两种类型：

1. 封闭式采访，即信息持有者回答一组预定的问题；
2. 开放式采访，即没有一个预先准备好的程序。需求工程团队即兴提问，因此能达到对他们想要什么有一个更深的了解。

在具体实践中，对信息持有者的采访这两种方式可能都会用到。我们会得到某些问题的回答，但是这些回答通常会引发其他问题，由此还会针对这些新问题展开比较自由的讨论。完全自由式的讨论很少奏效；绝大多数的采访需要由一些问题来开头，并保持整个讨论过程集中在所有开发的系统上。

通过采访来全面了解系统信息是一种很好的办法。在采访中我们可以了解信息持有者是如何工作的，他们是如何与系统进行交互的，以及他们对当前系统所面临的问题和困难等。人们很愿意谈论他们的工作，因而会很乐意接受采访。但是，对于应用领域需求，通过采访的方式是很难获得的。

很难从采访中导出领域知识，有以下两个原因：

1. 所有的应用专家都使用专业术语和行话。如果让他们不使用这些专用术语来谈论领域需求几乎是不可能的。他们通常精确微妙地使用这些专业术语，很容易让需求工程师误解。
2. 有些领域知识对于这些信息持有者来说是太熟悉了，以至于他们要么是难以找到合适的词语来解释，要么会不自觉地认为这些概念太基本，不值得去解释。例如，对于图书管理员，将所有书籍在入库前分类处理是不言而喻的事情，但是，这一点对采访者来说却不是显而易见的，因此，可能就会在需求中忽视掉。

采访对于导出机构需求和约束来说不是一项很有效的技术，因为，在机构中不同人之间存

在着一些很微妙的权力关系。公布出来的机构结构很少能真正反映机构中的真实的决策模式，被采访者不愿意对陌生人揭露机构中真实的一面。通常情况下，大部分人不愿去涉及影响需求的这些政治和机构因素。

有能力的采访者通常有两个特征：

1. 他们思想开放，对需求没有先入之见，而且很愿意听取信息持有者的意见。如果信息持有者提出意外的需求，他们愿意主动改变自己对系统原有的看法。
2. 他们会用基本问题、需求建议或者是在一个原型系统中共同工作来引导被采访者开始讨论。简单的提问，比如，“告诉我你需要什么？”，是不会得到什么有价值的情报的。绝大多数人都认为在一个指定的情景中谈论要比泛泛地讨论容易得多。

通过采访所获得的信息补充了来自描述业务进程或当前系统文档、用户观察以及其他手段所获得的信息。有时，除了系统文档以外的信息，采访可能是唯一的系统需求来源。然而，采访本身很容易漏掉一些基本信息，所以，我们应该结合其他需求导出技术来使用它。

4.5.3 脚本

通常，人们容易把事物与现实生活中的例子相联系，而不容易把事物与一个抽象描述联系起来。若把人如何与一个软件系统交互用一个脚本的方法来描述的话，人们就很容易理解并且评论它的好与坏。需求工程师从对场景的评论中得到信息，然后再将其形式化地表示出来。

当把细节加入一个概要需求描述中时，脚本（场景）可能特别有用。脚本是对交互实例片段的描述。每个脚本可能包含一个或多个可能的交互。人们研究出多种不同的脚本形式，它们能在不同的细节层次上提供不同类型的信息。

脚本开始于一个交互框架，在导出过程中，细节被逐渐增加，直到产生一个完整的交互描述。在绝大多数情况下，一个脚本可能包括以下内容：

1. 在脚本的开始部分有一个系统和用户期望的描述。
2. 一个关于标准事件流的描述。
3. 一个关于可能出错的位置以及如何处理错误的描述。
4. 有关其他可能在同一时间进行的活动的信息。
5. 在脚本完成后系统状态的描述。

基于脚本的需求导出需要与信息持有者共同识别出脚本并捕获这些脚本的细节。可以用文本来书写脚本，并附带图和屏幕截图等。还有一种较结构化的方法，如事件脚本法或用例法也可以使用。

作为一个简单的文本脚本的例子，让我们看看 MHC-PMS 系统用户是如何使用系统来输入数据的。这个脚本如表 4-14 所示。当一个新病人进入诊所后，分诊医生会创建一条记录，并把患者信息（姓名、年龄等）添加进去。然后护士会接待病人并收集医疗历史记录。接着病人与医生有一个初步的会诊，医生为其诊断，若合适并会为其推荐一套治疗方案。上述脚本显示了当医疗历史记录收集好后所要发生的事件。

4.5.4 用例

用例是一种需求发现技术，这种技术首先是在 Objectory 方法（Jacobson 等，1993）中引入的。现已成为统一建模语言的一个基本特征。一个最简单的形式是，一个用例识别在一个交互中所参与的所有角色并命名该交互类型。然后还会附加一些描述系统交互的额外信息。这些信息可能是文本描述或是一个或多个图形化模型，例如 UML 序列图或状态图。

初始假设:

病人来到分诊医生面前, 该医生已经在系统中创建了一条记录, 并收集了病人的个人信息 (姓名、地址、年龄等)。一名护士登录到系统中并正在收集治疗历史信息。

正常状态:

护士通过姓来查找病人, 如果同一姓氏有多个病人, 就要用到病人的名和出生日期来查找病人信息。

护士选择菜单选项来添加治疗历史记录。

接着护士按照一系列的系统提示来输入信息, 这些信息包括: 有关心理问题的其他就诊地点 (自由文本输入), 现在的状况 (护士从菜单中选择), 当前所服药物 (从菜单中选择), 过敏史 (自由输入) 和家庭生活 (表格)。

有哪些会出错:

病人的记录不存在或不能找到。护士应该创建一条新纪录并记录病人个人信息。

在菜单中没有合适的病人的状况或所服药物。护士应该能选择其他选项并可以输入自由文本来描述病人的状况或所服药物。

病人不能或不愿在医疗历史记录中提供信息。护士应该输入自由文本来记录该病人不能或不愿提供信息。系统应该打印出标准的缺项表格来叙述那些所缺信息可能意味着治疗会被限制或是推迟。这份表格应该由病人签字并交给病人。

其他活动:

当正在输入信息时, 其他职员只能查阅但不能编辑该记录。

完成的系统状态:

用户登录。将包括医疗历史记录在内的病人记录输入到数据库。记录被添加到系统日志中, 系统日志显示了此时间段的开始和结束时间以及相关的护士。

图 4-14 在 MHC-PMS 例子中收集治疗历史记录的脚本

用例通过一个高层用例图记录下来。用例的集合代表所有将会在系统需求中出现的交互。过程中的角色可能是人, 也可能是其他系统, 用人形图标来表示。每一个交互类用一个命名的椭圆来表示。用例和交互类之间用线相连。也可以选择把箭头添加在连线一端, 代表交互的开始方向。用例表示方法如图 4-15 所示, 该图是心理健康护理病人的信息系统的用例。

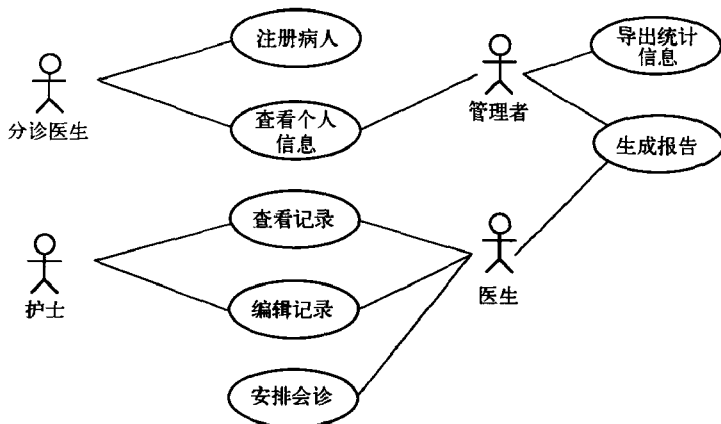


图 4-15 MHC-PMS 的用例

在脚本 (故事情景) 和用例之间没有硬性的区别。一些人认为一个用例就是一个单独的脚本, 如 Stevens 和 Pooley (2006) 所建议的, 一个用例封装了一组脚本, 每个脚本是用例中的一个单个线程。因此, 一个标准的交互就会有一个脚本, 每个可能的异常也会有脚本。在实践中, 我们可以用其中任何一种方法使用它们。

用例识别一个单个交互, 这是系统与用户之间或是系统与其他系统之间的交互。每一个用

例都应该用文本记载下来。然后会被链接到其他给出更详细脚本的 UML 模型。例如，可以这样简单描述图 4-15 中安排会诊的用例：

安排会诊允许两个或更多的医生，在不同的办公区的同一时间查看相同的记录。一个医生通过从医生的下拉菜单选择一个在线的医生来发起会诊。病人记录也会在其他人的屏幕上显示，但是只有发起会诊的那个医生可以编辑此记录。另外会创建一个文本聊天窗口来协调动作。假定一个音频电话会议会被单独建立。

对于从直接与系统交互的信息持有者处导出需求来说，脚本和用例是有效的技术。每一种交互类型可以表示为用例。然而，因为它们集中在与系统的交互上，它们对于导出约束或高层业务以及非功能需求抑或是领域需求发现都是不奏效的。

UML 对于面向对象模型来说是一个约定俗成的标准。因此用例和基于用例的导出现在被广泛应用于需求导出中。在第 5 章中会进一步讨论用例，并讲解如何将用例与其他系统模型一起用于系统设计的文档化。

4.5.5 深入实际

软件系统不是孤立存在的，它们是在一个社会和机构的环境中使用的，软件需求来自于这样的环境和受到这个环境的制约。满足这些社会的和机构的需求对系统的成功是至关重要的，许多软件系统被交付但是从未使用过，其中一个原因就是他们的需求没有正确地考虑到社会和机构是如何影响系统的实际运行的。

深入实际是一项能用来了解操作过程并帮助导出这些过程支持需求的观察技术。分析人员投身到待用系统的工作环境中，通过观察日常工作并记录参与者相关的实际任务。深入实际的价值是它能帮助发现隐性的系统需求，这些需求是反映了人们实际的工作方式，而不是机构所定义的规范化过程。

人们经常发现说清楚自己的工作细节是件非常困难的事，因为他们不善此道。他们了解自己的工作但可能不了解在机构中与其他工作之间的联系。社会的和机构的因素对工作的影响对于机构中的人员可能是很难了解到的，只有从一个没有偏见的观察者眼中才能看到这些。例如，一个工作组可以进行自我组织，这样组员就可以知道彼此的工作情况，如果某个人缺席还可以移交任务。这个在采访中也许不会被提及，因为该组不会把这视为工作中整体的一部分。

Suchman (Suchman, 1987) 用深入实际方法来研究办公室工作，他发现实际的工作场景具有丰富的内涵，与简单的被办公室自动化假定的模型相比要复杂和动态得多。假设的工作模型和真实工作的不同是造成这些办公系统没有对生产率产生重要影响的最重要原因。Crabtree (2003) 从那以后讨论了更广的研究领域，并大体上描述了深入实际在系统设计中的用法。在本书作者的个人研究中，调查了将深入实际方法与软件工程过程集成在一起的方法，主要通过两个手段：一是将深入实际方法与需求工程诸多方法联系起来使用 (Viller 和 Sommerville, 1999; Viller 和 Sommerville, 2000)，二是通过记录在协作系统中的交互模式 (Martine 等, 2001; Martin 等, 2002; Martin 和 Sommerville, 2004)。

深入实际对发现两种类型的需求特别有效：

1. 需求来自人们实际的工作方式而非过程定义中所要求的工作方式。举例来说，空中交通管制员可能关掉飞机冲突警报系统，该警报系统能检测飞机是否在交叉航线上。而按正常控制流程指定此警报系统是应该使用的。他们故意把飞机置于冲突航道中短暂时间，这有助于管理空域。他们设计控制策略用来确保飞机在问题发生之前已经离开，他们发现冲突警报分散了他们工作的注意力。

2. 需求来源于合作和对别人活动的了解。举例来说，空中交通管制员可能通过对其他管制

员工作的了解来预知将要飞入他们控制区域的飞机数量,然后他们按此修改控制策略。因此,一个自动化的 ATC 系统应该允许一区域管制员能看到相邻的区域的工作状况。

深入实际方法可以和原型法结合使用(见图 4-16)。深入实际法能告知开发原型,使原型开发中所需的精炼循环数减少。此外,原型法通过识别问题和疑问并和深入实际的人员一起讨论这些问题和疑问来执行深入实际法。他(或她)应该在系统研究的下个阶段寻找问题的答案(Sommerville 等,1993)。

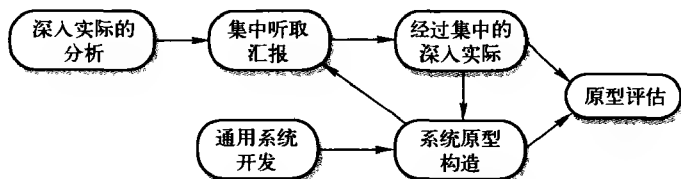


图 4-16 需求分析的深入实际方法和原型法

深入实际研究能暴露关键过程的细节,这些细节时常被其他需求导出技术所遗漏。然而,因为它集中在最终用户身上,所以这个方法不总是适合发现机构或领域的需求。它不能够总是识别出系统应该添加的新特征。因而,深入实际不是一个完全的需求导出方法,它应该和其他方法结合使用,如用例分析。



需求复查

需求复查是个过程,一组来自系统客户和系统开发方面的人员详细地阅读需求文档并检查其中的错误、异常和不一致性。一旦检查出任何问题并记录下来,接着客户就需要和开发人员协商如何解决所发现的问题。

<http://www.SoftwareEngineering-9.com/Web/Requirements/Reviews.html>

4.6 需求有效性验证

需求有效性验证是检验需求是否真正按客户的意愿来定义系统的过程。它和需求分析有很多共性,都是要发现需求中的问题。需求有效性验证很重要,如果在后续的开发或当系统投入使用时才发现需求文档中的错误,就会导致更大代价的返工。

由需求问题而对系统做出变更的成本比修改设计或代码错误的成本要大得多。理由是,需求的变更总是带来系统设计和实现的变更,从而使系统必须重新测试。

在需求有效性验证过程中,要对需求文档中定义的需求执行多种类型的检查。这些检查包括:

1. **有效性检查** 某个用户可能认为系统应该执行某项功能。然而,进一步的思考和分析可能发现还需要添加额外的功能,或是发现系统需要的是不同的功能。系统有很多不同的用户,而这些用户可能需要不同的功能,因此,任何一组需求都不可避免地要在不同用户之间协商。

2. **一致性检查** 在文档中,需求不应该彼此冲突。即对同一个系统功能不应出现不同的描述或相互矛盾的约束。

3. **完备性检查** 需求文档应该包括所有系统用户想要的功能和约束。

4. **真实性检查** 基于对已有技术的了解,检查需求以保证需求能真正实现。这些检查还要考虑到系统开发的预算和进度安排。

5. 可检验性检查 为了减少在客户和开发商之间可能的争议，系统需求的书写应该总是使之是可以检验的。这意味着能设计出一组检查方法来验证交付的系统是否满足每一个定义的需求。

有许多能联合使用或单独使用的需求有效性验证技术，主要有：

- 1. 需求评审** 由一组评审人员对需求进行系统性分析，主要是错误检查和不一致性检查。
- 2. 原型建立** 在这个有效性验证方法中，为客户和最终用户提供一个可执行的系统模型。他们能在这个模型上体验从而检查系统是否符合他们的真正需要。
- 3. 测试用例生成** 需求应该是可测试的。如果将把对需求的测试作为有效性验证过程的一部分，经常会发现需求中很多问题。如果一个测试的设计很困难或是不可能的，那通常意味着需求的实现将会很困难，应该对此重新考虑。在编写任何代码之前就对用户需求开发测试用例是极限编程的一个完整部分。

需求有效性验证中产生的问题不应该被低估。最终论证一组需求实际上是否符合了用户的需要是很困难的。用户需要勾画出系统的操作过程并构想如何让系统符合他们的工作。即使是一个有经验的计算机专家，做这种抽象的分析工作都是相当困难的，更何况是系统用户了。因此，需求的有效性验证不可能发现所有的需求问题，在需求文档被确认之后，不可避免会有进一步的需求变更来更正遗漏和误解。

4.7 需求管理

大型软件系统的需求总是在变化的。原因之一是通常要开发这些系统来满足棘手的问题，这些问题不可能被完全定义。因为问题没有被完全定义，所以软件需求注定是不完全的。在软件过程中，信息持有者对问题的理解是在不断变化的（见图 4-17），因而系统需求必须要相应地反映这些对问题观点的改变。

一个系统一旦被安装并被正式使用，就会不可避免的会产生新的需求。对于用户和系统客户来说，很难预料新系统将会对他们的业务过程和工作方法产生什么影响。一旦最终用户对系统有了经验，他们就会发现新的需求和更重要的需求。变更不可避免会出现的原因有以下几个：

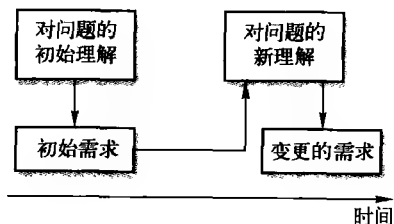


图 4-17 需求进化

1. 系统业务和技术环境在安装之后总是在发生变化。新的硬件可能被引进，系统与其他系统的接口是必需的，业务优先次序可能会改变（并伴随着所要求的系统支持的变化），新的立法和规章制度的实行使得系统必须做相应的调整。

2. 系统购买者和系统最终用户很少是同一人。系统客户可能因为机构原因或预算约束对系统提出一些需求，而这些需求可能和最终用户需求冲突。而且在交付之后，如果系统要满足目标要求，就必须加入新的功能。

3. 大型系统通常拥有不同的用户群。不同的用户有不同的需求和优先次序。这些可能是冲突的或是矛盾的。最终的系统需求不可避免要在他们之间进行折中，随着经验的积累，对不同用户支持上的这种平衡需要改变。

需求管理是一个对系统需求变更了解和控制的过程。我们需要跟踪各个需求并维护有依赖关系的需求之间的联系，这样我们就能够评估需求变更的影响。我们需要建立一个形式化的过程来形成变更建议，并将它们与系统需求联系起来。一旦形成了需求文档的草稿版本，需求管理的形式化过程就应该开始。然而，我们也应该开始规划在需求导出过程中如何管理需求变更。



持久性和易变性需求

某些需求较之其他需求更容易改变。持久性需求是那些与核心的、变化很缓慢的机构活动相关联的一类需求。持久性需求与基础性的工作活动有关。易变性需求是更容易改变的一类需求。它们总是与那些反映机构如何去做它们的工作而不是工作本身的一些支持性活动相关。

<http://www.SoftwareEngineering-9.com/Web/Requirements/EnduringReq.html>

4.7.1 需求管理规划

规划是需求管理过程中第一个重要阶段。规划阶段要确定需求管理所需的细节水平。在需求管理阶段，必须决定以下内容：

1. **需求识别** 每个需求要有一个唯一的标识以便可以被其他需求交叉索引，同时可以用到可追溯的评估中。

2. **变更管理过程** 这是一组对变更带来的影响和成本进行评估的活动。这在紧接的一节中对此做更详细的讨论。

3. **可追溯策略** 这些策略定义了需求之间的关系以及需求和系统设计之间的关系，这些关系是要被记录下来的，对此要给出记录的维护方法。

4. **工具支持** 需求管理涉及对大量需求信息的加工。可以使用的工具的范围很广，从专家需求管理系统到电子表格和简单的数据库系统。

需求管理需要一些自动化手段的支持，在规划阶段要对使用的软件工具做出选择。需要用工具支持的地方有：

1. **需求存储** 需求的存储应该是安全的、可管理的，在需求工程过程中的每个人对此都是可以访问的。

2. **变更管理** 变更管理（见图4-18）的过程若是由有效的工具来支持将变得很简单。

3. **可追溯性管理** 依照上面的讨论，对可追溯性的工具支持能发现相关的需求。一些使用自然语言处理技术的工具能发现需求之间可能的关联。

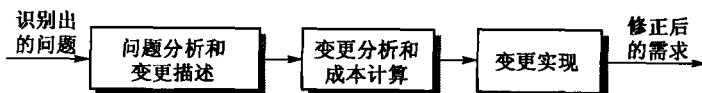


图4-18 需求变更管理

对于小型系统，可能不必使用专业化的需求管理工具。需求管理过程用字处理器中的工具、电子表格和微机数据库就能支持。然而，对比较大的系统，更多专业化工具的支持是必需的。本书的网页上有关于需求管理工具的相关信息的链接。



需求可追溯性

你需要跟踪需求、它们的来源以及系统设计之间的关系，这样你就能分析所提议的变更请求的原因以及变更会对系统其他部分造成的影响。你需要有追踪变更是如何在系统中扩散的能力。为什么？

<http://www.SoftwareEngineering-9.com/Web/Requirements/ReqTraceability.html>

4.7.2 需求变更管理

需求变更管理（见图4-18）应该应用到需求文档被确认之后对系统需求的所有变更提议。变更管理是极其关键的，因为你需要决定是否实现一个新需求所带来的利益相对于实现它所需要的付出是合理的。在变更管理中使用形式化过程的好处是，所有的变更提议都被一致地处理，而且对需求文档的变更是在一种可控的方式下进行的。

一个变更管理过程有3个基本阶段：

1. 问题分析和变更描述 该过程始于一个识别出来的需求问题或是一份详尽的变更提议。在这个阶段，要对问题或变更提议进行分析以检查它的有效性。分析结果反馈给变更请求者，请求者进而产生一个更加详尽的需求变更提议或是取消该变更请求。

2. 变更分析和成本计算 使用可追溯性信息和系统需求的一般知识对被提议的变更所产生的影响进行评估。变更的成本估算包括对需求文档的修改，而且在适当的时候，还包括系统设计和实现。一旦分析完成，就有了对此需求变更是否执行的决策意见。

3. 变更实现 必要的话，需求文档以及系统设计和实现都要做修改。要对需求文档进行组织，使得变更不会带来大量文字的重写和重组。与程序类似，文档的可追溯性是通过最小化外部引用和尽量使之模块化来实现的。这样，就可以修改和代替个别部分而不会影响文档的其他部分。

如果必须要立即实现一个新需求，总是有先对系统做变更然后再回头修改需求文档的想法。应该尽力避免这种情况，因为这几乎不可避免地导致需求描述和系统实现不同步。一旦系统变更完成，很容易忘记变更需求文档或忘记往需求文档里添加信息，只有相应的文档信息得到更新才能保证与实现是一致的。

人们设计了敏捷开发过程，例如极限编程，以应对在开发过程中变更需求。在这些过程中，当某个用户提出一个需求变更时，并不通过一个形式化的变更管理过程，而是用户必须对变更设置优先权，如果它有一个很高的优先权的话，我们就要决定为下个迭代所规划的哪些系统特征应该被取代。

要点

- 一个软件系统的需求描述了系统应该做什么以及定义系统运行时和实现时的约束。
- 功能需求是有关系统一定要提供的服务或者是必须执行的计算的描述。
- 非功能需求约束所开发的系统和所采用的开发过程。它们可能是产品需求、机构需求或者是外部需求。这些需求与系统的总体特性相关，且作用于系统整体。
- 软件需求文档是经过认可的系统需求描述。它应该被很好地组织，以便既可以为系统客户使用又可以为软件开发者使用。
- 需求工程过程包括可行性研究，需求导出和分析，需求描述，需求有效性验证和需求管理。
- 需求的导出和分析是一个反复的过程，可以看做是一种螺旋式活动——需求发现，需求分类和组织，需求协商和需求文档化。
- 需求有效性验证是检查需求的有效性、一致性、完备性、真实性和可检验性。
- 业务、组织机构和技术上的变更不可避免地导致软件系统需求的变更。需求管理就是管理和控制这些变更的过程。

进一步阅读材料

《Software Requirement, 2nd ed.》这是一本为编写和使用需求的人设计的书。它讨论了好的需求工程实践 (K. M. Weigers, 2003, Microsoft Press)。

《Integrated requirements engineering: A tutorial》这是作者写的一篇论文, 讨论了需求工程活动和这些活动如何与现代软件工程实践相适应 (I. Sommerville, IEEE Software, 22 (1), Jan-Feb 2005)。http: //dx. doi. org/10. 1109/MS. 2005. 13。

《Master the Requirements Process》这是一本基于特别方法 (VOLERE) 同时也包括很多好的关于需求工程一般性建议的书, 写得很好, 而且容易阅读 (S. Robertson and J. Robertson, 2006, Addison-Wesley)。

《Research Directions in Requirements Engineering》这是关于需求工程研究的一份很不错的调查, 突出了未来该领域研究中的挑战, 提出了像规模和敏捷这样的问题 (B. H. C. Cheng and J. M. Atlee, Proc. Conf on Future of Software Engineering, IEEE Computer society, 2007)。http: //dx. doi. org/10. 1109/FOSE. 2007. 17。

练习

- 4.1 找出对基于计算机的系统的4种可能要定义的需求, 并简要地描述之。
- 4.2 在一票务系统的部分需求描述中有下面一段, 请找出其中的二义语句和遗漏的地方:
自动票务系统能够发售火车票。用户选择他们的目的地并输入信用卡以及个人身份证号。火车票就输出了, 同时相应费用从信用卡划出。当用户按下开始按钮, 关于可能的目的地菜单就显示出来了, 同时有让用户选择目的地的消息给用户。一旦选择了目的地, 系统就请求用户输入他们的信用卡。系统检验卡的有效性, 然后请求用户输入个人身份证号。当信用交易得到有效性验证, 系统就开始出票了。
- 4.3 使用在本章中给出的结构化方法重写上述的描述, 以一些适当的方式解决被识别的二义性。
- 4.4 为上面提到的票务系统写出一组非功能需求, 指定对它预期的可靠性和反应时间。
- 4.5 使用这里建议的标准方式的自然语言技术, 为下面的功能写用户需求:
 - 一个无人值守的汽油 (燃料) 泵系统, 包括一个信用卡读卡机。客户刷卡后指定所要的数量, 燃料开始传输, 费用从客户账户划出。
 - 银行自动柜员机中的现金分发功能。
 - 字处理器中的拼写检查和改正功能。
- 4.6 对于负责提取系统需求描述的工程人员, 如何搞清功能需求和非功能需求之间的关系? 给出你的建议。
- 4.7 利用你关于自动取款机的使用知识, 设计一组用例, 用来作为理解自动取款机系统需求的基础。
- 4.8 应该有哪些人参与需求复查? 画出一个过程模型来说明需求复查应该如何组织。
- 4.9 当系统必须要做紧急变更时, 可能要在批准需求变更之前对系统软件做出修改。给出一个执行这些修改的过程模型, 要保证需求文档和系统实现的一致性。
- 4.10 假设你刚在软件客户机构中谋得一份工作, 该软件客户又正好与你先前的老板签订了一份系统开发合同, 你可能发现你现在的公司和先前的公司对需求有不同的解释。讨论在这样的情形中你应该怎样做。你知道如果二义性不被解决, 你目前雇主的成本将会增加。你也有对先前雇主保密的承诺。

参考书目

- Beck, K. (1999). 'Embracing Change with Extreme Programming'. *IEEE Computer*, **32** (10), 70–8.
- Crabtree, A. (2003). *Designing Collaborative Systems: A Practical Guide to Ethnography*. London: Springer-Verlag.
- Davis, A. M. (1993). *Software Requirements: Objects, Functions and States*. Englewood Cliffs, NJ: Prentice Hall.
- IEEE. (1998). 'IEEE Recommended Practice for Software Requirements Specifications'. In *IEEE Software Engineering Standards Collection*. Los Alamitos, Ca.: IEEE Computer Society Press.
- Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. (1993). *Object-Oriented Software Engineering*. Wokingham: Addison-Wesley.
- Kotonya, G. and Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques*. Chichester, UK: John Wiley and Sons.
- Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Englewood Cliff, NJ: Prentice Hall.
- Martin, D., Rodden, T., Rouncefield, M., Sommerville, I. and Viller, S. (2001). 'Finding Patterns in the Fieldwork'. *Proc. ECSCW'01*. Bonn: Kluwer. 39–58.
- Martin, D., Rouncefield, M. and Sommerville, I. (2002). 'Applying patterns of interaction to work (re)design: E-government and planning'. *Proc. ACM CHI'2002*, ACM Press. 235–42.
- Martin, D. and Sommerville, I. (2004). 'Patterns of interaction: Linking ethnomethodology and design'. *ACM Trans. on Computer-Human Interaction*, **11** (1), 59–89.
- Robertson, S. and Robertson, J. (1999). *Mastering the Requirements Process*. Harlow, UK: Addison-Wesley.
- Sommerville, I., Rodden, T., Sawyer, P., Bentley, R. and Twidale, M. (1993). 'Integrating ethnography into the requirements engineering process'. *Proc. RE'93*, San Diego CA.: IEEE Computer Society Press. 165–73.
- Stevens, P. and Pooley, R. (2006). *Using UML: Software Engineering with Objects and Components, 2nd ed.* Harlow, UK: Addison Wesley.
- Suchman, L. (1987). *Plans and Situated Actions*. Cambridge: Cambridge University Press.
- Viller, S. and Sommerville, I. (1999). 'Coherence: An Approach to Representing Ethnographic Analyses in Systems Design'. *Human-Computer Interaction*, **14** (1 & 2), 9–41.
- Viller, S. and Sommerville, I. (2000). 'Ethnographically informed analysis for software engineers'. *Int. J. of Human-Computer Studies*, **53** (1), 169–96.

系统建模

目标

本章将介绍一些不同类型的系统模型，这些模型是在需求工程和系统设计过程中必须建立的模型。读完本章，你将了解以下内容：

- 理解如何用图形模型来表示软件系统；
- 理解为什么需要不同类型的模型以及基本的系统建模角度，如上下文、交互、结构、行为等；
- 介绍统一建模语言（UML）中定义的一些图表类型，以及如何在系统建模中使用这些图；
- 理解模型驱动工程中包含的思想，从结构模型和行为模型自动地生成系统。

系统建模就是建立系统抽象模型的过程，每一个模型表示一个系统不同的角度或方面。系统建模通常意味着用一些图形符号表示系统，现在几乎都是用统一建模语言（UML）中的符号。然而，也有可能要建立系统的形式化（数学）模型，其通常作为详细的系统描述。本章中讲述使用 UML 的图形建模，在第 12 章会讲述形式化建模。

在需求工程过程中使用模型是为了帮助我们导出系统的需求，在设计过程中使用模型是为了向实现系统的工程师描述系统，并且实现系统之后还要用模型来论证系统的结构和操作。我们可能要建立现有系统和将要开发系统两个系统的模型。

1. 在需求工程中要用到现有系统的模型。它们给我们阐释现有系统的功能，并作为讨论系统功能扩展和不足的基础。这些都会导出新系统的需求。

2. 在需求工程中使用新系统的模型有助于解释其他系统信息持有者提出的需求。工程师使用这些模型来讨论设计想法，并为系统实现进行论证。在模型驱动工程过程中，由一个模型有可能实现一个完整的系统，也有可能只是实现系统的一部分。

系统模型的很重要的一个方面是在于它摆脱了所有的细节内容。系统模型是系统的一个抽象而不是它的一种替代表现。理想情形，系统的一种表示应该保持所要表示的实体的所有信息。一个抽象是千方百计进行简化，以便找出最突出的特征。例如，举一个不很恰当的比喻，如果将作者的这本书在报纸上连载，那刊登出来的应该是本书要点的一个摘要内容，如果将本书从英语翻译成意大利语版本，那它就是一个替代表现，翻译者的意图是将英语版的所有内容都原封不动地维持过来。这些模型可以从不同的角度去表述系统：

1. 从外部来看，它是对系统上下文或系统环境建模。
2. 从交互上看，它是对系统与环境之间或是一个系统各组成部分之间的交互建模。
3. 从结构上看，它是对系统的体系结构和系统处理的数据的结构建模。
4. 从行为上看，它是对系统的动态行为和它对事件的响应方式建模。

这些方面与 Kruchten 提出的系统体系结构的 4 + 1 角度（Kruchten, 1995）有很多相似之处，Kruchten 认为我们应该从不同方面来论证系统的体系结构。在第 6 章中我们将会来讨论这种 4 + 1 方法。

在本章中，使用 UML（Booch 等，2005；Rumbaugh 等，2004）中定义的图，UML 已经成为面向对象建模的标准建模语言。UML 中有许多类型的图，因此可以建立许多不同类型的系统模

型。然而,在2007年(Erickson和Siau,2007)做的一个调查表明:UML的大部分使用者认为5种类型的图就可以表示出一个系统的本质:

1. 活动图,它表示一个过程或数据处理中所涉及的活动。
2. 用例图,它表示一个系统和它所处环境之间的交互。
3. 时序图,它表示参与者和系统之间以及系统各部分之间的交互。
4. 类图,它表示系统中的对象类以及这些类之间的联系。
5. 状态图,它表示系统是如何响应内部和外部事件的。

由于在这里不能讨论UML中的所有图类型,那我们就来重点看一下在系统建模中主要使用的这5种主要类型的图。

在系统建模时,图形符号的使用通常是非常灵活的,不必严格坚持符号的细节。一个模型的具体程度和严格程度取决于我们想怎样来使用系统模型。图形建模通常有3种用途:

1. 为讨论现有系统或提出新功能的系统提供便利。
2. 论证现有系统。
3. 作为详细的系统描述,该描述可用来产生系统实现。

第一种情况下,模型的目的是促进开发系统的工程师之间的讨论,这些模型可能并不完整(只需涵盖讨论的要点即可),他们使用的符号可能也不正式。这就是模型在所谓的“敏捷建模”中通常的使用方式(Ambler和Jeffries,2002)。当使用模型来进行论证时,所建模型不必是完整的,因为我们可能只想建立一个系统的一些部分的模型。然而,这些模型必须是正确的,它们必须正确地使用符号,并作为系统的准确描述。



统一建模语言

统一建模语言中有13种不同类型的图,它们用来为软件系统建模。它是20世纪90年代面向对象建模的成果,是由相似的面向对象的符号集合集成而得的,在2004年经过一次较大的修订之后被确定了下来,也就是UML2。UML通常被认为是软件系统建模的标准语言。为了支持更一般的系统建模,有人提出了不同的建议。

<http://www.SoftwareEngineering-9.com/Web/UML/>

在第三种情况下,模型是作为以模型为基础的开发过程的一个组成部分,系统模型必须是完整且正确的。这是因为它们是产生系统源代码的基础。因此,我们必须非常认真以避免混淆类似的符号,比如stick和block arrowhead所表示的意思是不同的。

5.1 上下文模型

在系统描述的早期阶段,应该首先界定系统的边界,与系统信息持有者一起明确系统应具备什么样的功能以及系统环境提供了什么。我们应该确保系统实现对一些业务过程的自动支持,但是其他的应该是手工过程或是由不同的系统支持。我们应该查看现有系统在功能上可能的重合部分,并决定应该在哪里实现新功能。我们应该在过程的早期阶段就完成这些判断,以便限制系统成本以及分析系统需求和设计中所需要花费的时间。

在许多情况下,系统与环境间的界限是相对清楚的,例如,当准备用一个自动化系统去取代现有的手工劳动或是半自动化系统时,新系统的环境与旧系统的环境是一样的。在另外一些情况下,却有许多不确定性,需要在需求分析过程中不断认识。

例如,我们开发一个心理治疗信息管理系统。这个系统用来管理来心理诊所进行心理治疗

的病人以及治疗的信息。在开发该系统的描述时，必须明确这个系统是否只用来收集咨询信息的（用其他系统收集病人的个人信息），还是也收集病人的个人信息。用其他系统管理病人信息的好处是可以避免信息的重复。然而，这样做最大的不便，可能会降低访问信息的速度。如果这些系统不能被访问，那么就不能使用 MHC-PMS 了。

定义系统的边界是有意义的，由于社会和机构的原因，系统边界的确定充满了非技术性因素。比如，必须精确定位系统的边界，使得分析过程可以完全在一个地点完成；也可能需要使系统分析避开与某个难以相处的管理者的沟通；或者使得成本增长、部门必须扩张才能完成系统的设计和实现。

系统边界一旦确定，接下来的部分分析活动就是定义系统上下文和系统与环境之间的依赖关系，一般而言，在这个活动中，第一步是建立一个简单的体系结构模型。

图 5-1 是一个简单的上下文模型，它描述了病人信息系统以及它所处环境中的其他几个系统。从图 5-1 可以看出 MHC-PMS 与一个预约系统和一个更一般的病人记录系统相连以共享信息。此外，这个系统还与管理报告系统、医院床位分配系统以及收集信息以供研究的系统相连。最后，利用一个处方系统生成病人用药的处方。

上下文模型通常表示某一环境包括几个其他的自动系统，然而它们却没有描述其他子系统之间的以及待描述的系统与它们之间的关联关系的类型。外部系统可能产生数据供该系统使用，同时也使用该系统生成的数据。这些周边子系统可能与该系统共享数据，可能与系统直接相连，或通过网络相连，或者根本就不连在一起。在空间上，这些子系统可能与该系统同在一处，也可能分处在不同的建筑物中，所有这些因素都将影响系统的需求和设计，必须加以考虑。

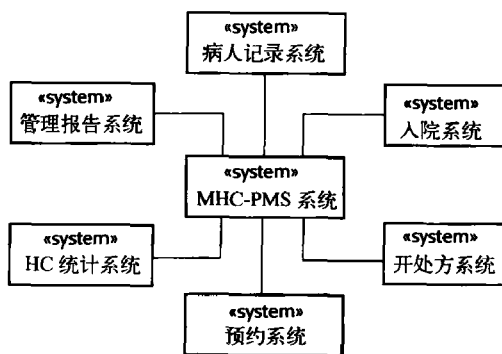


图 5-1 MHC-PMS 系统的上下文

因此，简单的上下文模型要同其他模型结合在一起使用，比如业务过程模型。它们描述使用了特有的软件系统的人工或自动化的处理过程。

图 5-2 是一个重要的系统过程的模型，它展示了 MHC-PMS 的应用过程。有时，患有心理健康问题的病人对别人或自己都有可能造成危害。因此他们必须被强制隔离在医院以获得治疗。这样的隔离是受严格的法律保护的。比如，必须定期重新考虑是否对病人继续进行隔离以免其被无限期地隔离。MHC-PMS 的其中一个功能就是确保实现这样的保护。

图 5-2 是一个 UML 活动图。活动图用来表示组成一个系统过程的活动并控制活动间的控制流。过程开始用实心圆表示，过程结束用套在另一个圆中的实心圆表示。圆边矩形表示活动，也就是必须得到执行的一个子过程。在活动图中可以包括对象，图 5-2 给出了可以用来支持不同过程的系统，它们是分离的系统，使用 UML 固有特性。

在 UML 活动图中，箭头表示各活动间的工作流，实心条表示各活动间的协调关系，当来自多个活动的流导向一个实心条时，那么只有这些活动均完成时过程才能进行。当来自实心条的流导向多个活动时，这些活动应同时执行。因此，在图 5-2 中，通知社会监护部门和病人最近亲属以及更新注册 3 个活动是同步的。

在箭头旁边可用条件注释来说明流的发生条件。在图 5-2 中，就是使用注释表示对社会构成威胁和没有构成威胁的两种病人的不同流向情况。对社会构成威胁的病人必须被隔离在安全场所。然而，那些有自杀倾向的、即对他们自己构成威胁的病人就应该被隔离在医院中合适的病房里。

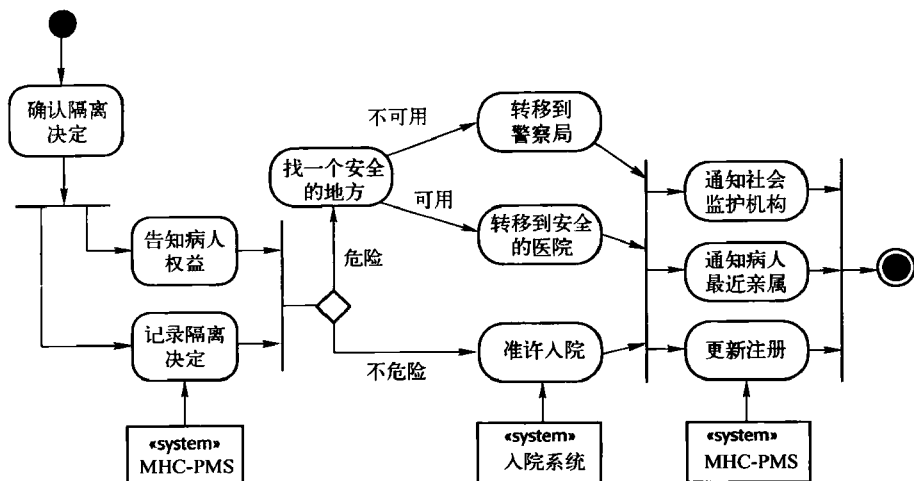


图 5-2 非自愿隔离的过程模型

5.2 交互模型

所有系统都会涉及一些交互。有可能是用户交互，与用户输入输出有关；有可能是正在开发的系统与其他系统之间的或是系统各部分之间的交互。为用户交互建模主要是因为它有助于我们识别用户需求。为系统与系统间的交互建模应将重点放在可能产生的交流问题上。为系统各部分之间的交互建模有助于我们分析所提出的系统结构能否实现系统所需的功能及其可靠性。

在本节中，将讲述两种相关的交互建模的方法：

1. 用例建模，该方法主要用来为系统与外部参与者（用户或其他系统）之间的交互建模。
2. 时序图，该方法用来为系统各部分之间的交互建模，尽管也包括一些外部因素。

用例模型和时序图在不同细节程度上表示交互，因此可能在一起使用。高水平用例所表示的交互的细节可能在时序图中得到刻画。UML 中的交流图也可以对交互建模，但因为它们是时序图的不同表示形式，所以在这里就不讨论交流图了。实际上，可利用一些工具将时序图生成交流图。

5.2.1 用例建模

用例建模最早是在 20 世纪 90 年代由 Jacobson 等（1993）所提出并收录在 UML 第 1 版（Rumbaugh 等，1999）中的。在第 4 章中已经知道，用例建模被广泛用于支持需求导出。一个用例可以被当做一个简单的脚本（情景），用来描述用户对系统功能的期望。

每一个用例表示一个具体的任务，涉及与系统的外部交互。在其最简单的形式中，用例用一个椭圆表示，参与者用一个小人儿表示。图 5-3 表示了 MHC-PMS 中的一个用例，它代表的任务是将 MHC-PMS 中的数据上传给一个更通用的病人信息记录系统。这个更通用的系统仅仅存储病人的简单数据信息，而不包含每一次咨询的信息，后者记录在 MHC-PMS 中。

可以看出在这个用例中有两个参与者：传递数据的操作员和病人信息记录系统。这种小人最早是用来表示人与人之间的交互的，但是现在也被用来表示其他外部系统和硬件。正常情况下，UML 中的用例图用不带箭头的直线表示信息流动方向，很明显，用例中的信息是双向传递的。然而，图 5-3 中随意

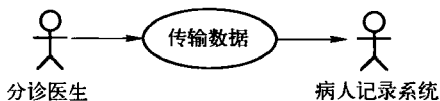


图 5-3 传输数据的用例

地使用箭头，这是为了表示分诊医生启动了此事务并且数据是传向病人信息记录系统的。

用例图所给出的只是交互非常简单的概要，所以我们必须提供更多的细节才能理解交互的内容。这些细节或许是一个简单的文本描述，或者是一个表格形式的结构化描述，或许是下面即将讨论的时序图。我们应该根据用例，即根据用例本身和所需的细节层次，选择最为合适的格式。有一个标准的表格格式使用起来最为便利，图 5-4 给出了“传输数据”用例的此种表格描述。

MHC-PMS: 传输数据	
参与者	分诊医生, 病人记录系统 (PRS)
描述	分诊医生会将数据从 MHC-PMS 系统传递给更通用的病人记录数据库, 该数据库由卫生主管部门维护。信息传递也许是更新个人信息 (地址、电话等), 也许是简短记录病人的诊断和诊治
数据	病人的个人信息, 治疗摘要
激励	由分诊医生发出的用户指令
响应	来自关于 PRS 已经更新的确认信息
注释	分诊医生必须具有适当的信息安全许可以便访问病人的信息和 PRS 系统

图 5-4 用例“传输数据”的表格描述

在第 4 章中已经知道，复合用例可以表示许多不同的用例。有时，可能仅用一个复合用例图就可以包括一个系统所有可能的交互。然而，由于用例数量可能很多，一个复合用例图可能无法包含系统中所有的交互。在这种情况下，就需要建立几张图，每一张表示几个相关的用例。比如，图 5-5 表示了 MHC-PMS 中的所有用例，其中包括参与者“分诊医生”。

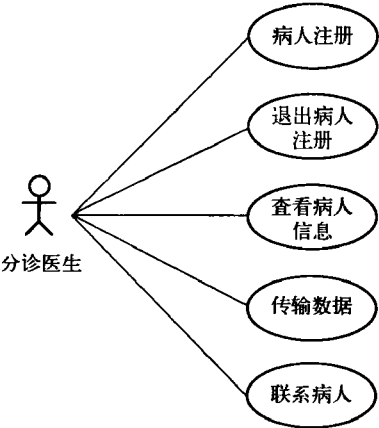


图 5-5 “分诊医生”角色的用例

5.2.2 时序图

UML 中的时序图最初是用来为参与者与系统对象之间的交互和对象与对象之间的交互建模的。UML 中的时序图的语法非常丰富，因此时序图可以对许多不同种类的交互建模。由于不能对所有可能性加以阐述，在这里就着重讲述一下时序图的基础。

顾名思义，时序图表示在特定用例中的交互发生顺序。图 5-6 是一个时序图的例子，它体现了这种符号的基本内容。这个图为显示病人信息用例中所涉及的交互建模，在用例中分诊医生可以看到一些病人信息。

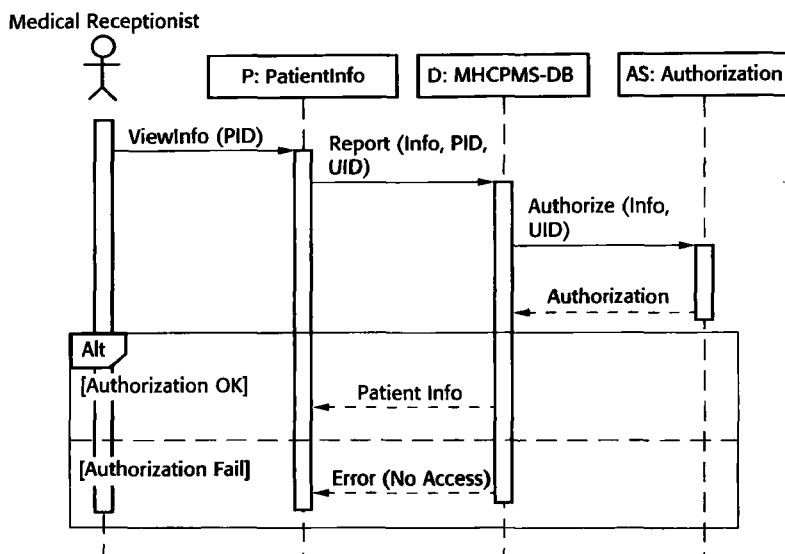


图 5-6 查看病人信息系统的时序图

涉及的对象和参与者列在图表顶端，向下垂直画一条虚线。对象之间的交互用带注释的箭头表示。虚线上的矩形表示相关对象的生命线（比如对象实例运行所需时间），从上向下为交互的顺序。箭头上的注释表示对对象的调用、它们的参数，以及返回值。在这个例子中，还使用了表示选择的符号。名为 alt 的框在方括号中所表示的条件下使用。

读者可以照下面顺序阅读图 5-6：

1. 分诊医生触发对象类 PatientInfo 的对象实例 P 中的 ViewInfo 方法，提供病人的标识符 PID。P 是用户接口对象，它以一个表格形式表示病人信息。
2. 对象实例 P 调用数据库返回所需信息。提供分诊医生的标识符以允许进行安全测试（在此阶段，我们无须关心 UID 的由来）。
3. 数据库使用某权限系统测试用户是否具备执行此动作的权限。
4. 如果有此权限，返回病人信息并以填充表格的形式显示在用户屏幕上。否则返回错误信息提示。

图 5-7 又给出了一个时序图的例子，这个时序图与图 5-6 中的时序图描述的是同一个系统，不过在其基础上增加了两项特征，也就是系统中参与者间的直接通信并创建对象作为操作序列中的一部分。此例中，创建了一个 Summary 类的对象来存储即将上传给病人信息记录系统的摘要数据。我们可以按下面的顺序阅读图 5-7：

1. 分诊医生登录病人信息记录系统。
2. 有两个选择可供选取。将更新的病人信息直接传递给 PRS，或将 MHC-PMS 中的健康摘要数据传递给 PRS。
3. 在每一个事例中，都会用权限系统检测分诊医生的权限。
4. 用户接口对象中的私人信息可以直接传递给 PRS，或者先在数据库中建立一项摘要记录，然后将此记录传递给 PRS。
5. 传递一完成，PRS 会发出一个状态信息，接着用户与系统断开。

除非我们要用时序图来生成代码或建立详细文档，否则不必在图中涵盖所有的交互。如果你在开发过程初期建立系统模型来支持需求工程和高层设计，那么依据具体的实现要求将会有许多的交互。比如，在图 5-7 中决定如何用用户 ID 来检测权限这个问题就是可以向后推迟的。

在实现过程中，这可能涉及与用户对象的交互，但这在此阶段并不重要，所以就不需要包括在此时序图中了。

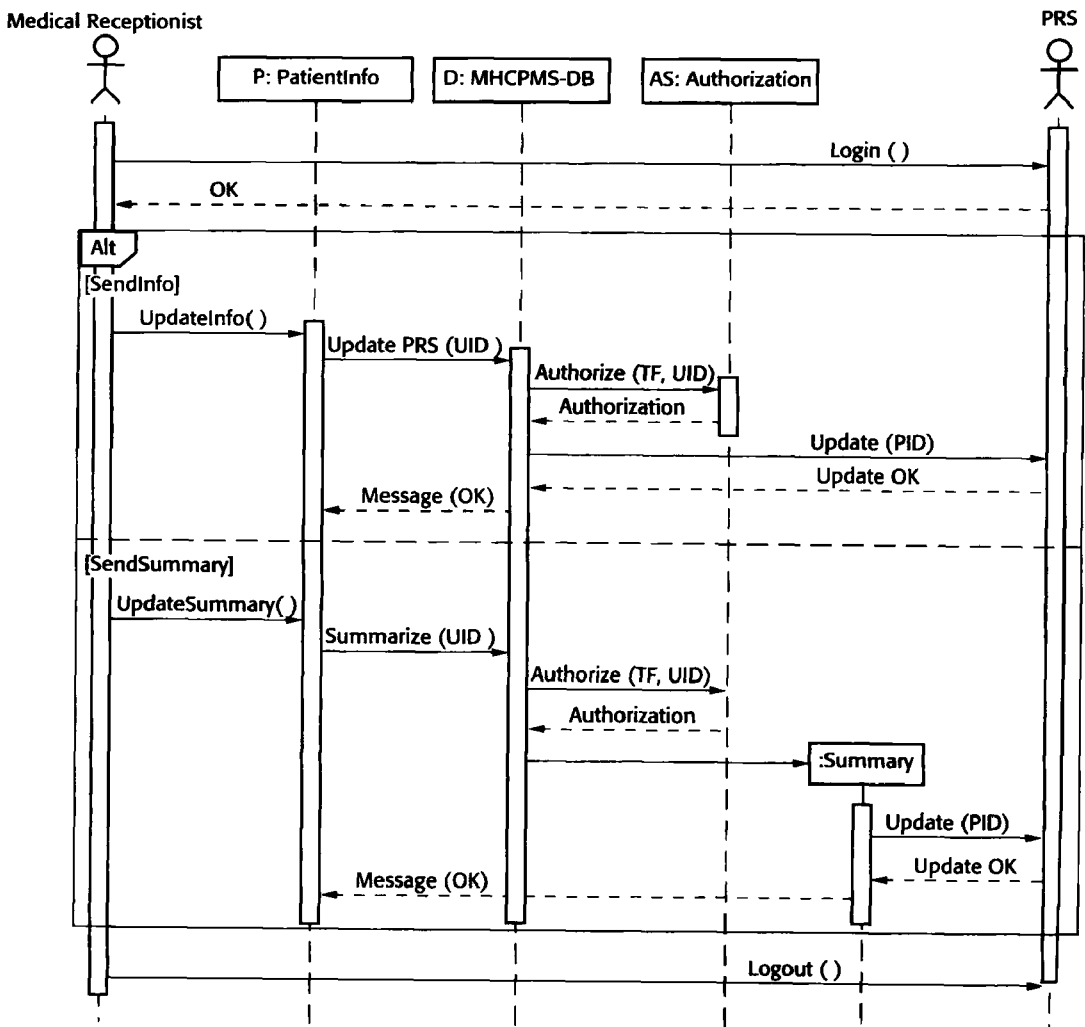


图 5-7 传输数据的时序图



面向对象的需求分析

在面向对象的需求分析过程中，使用对象类对现实世界中的实体建模。我们有可能建立多个不同类型的对象模型，有的用来表示对象类之间是如何联系的，有的用来表示如何将对象聚合在一起产生其他对象，有的用来表示对象与对象是如何交互的，等等。每一个模型都表示与所描述系统的独特信息。

<http://www.SoftwareEngingeeing-9.com/Web/OORA/>

5.3 结构模型

软件的结构模型表示的是系统的构成，表示为组件构成系统以及组件之间的关系。有的结构模型是静态模型，表示系统设计的结构；有的是动态模型，表示系统执行时的构成。这两者是

不同的。系统的动态结构是交互线程的集合，与描述系统组成部分的静态模型有很大的不同。

我们建立结构模型以讨论和设计系统体系结构。系统的体系结构的设计是软件工程中的一个非常重要的问题。当建立体系结构模型时，UML 组件包和部署图都可能用到。在第 6、18、19 章中将讲述软件体系结构的不同方面和软件体系结构建模。在本节，主要讲述类图在软件体系中对对象类的静态结构建模方面的应用。

5.3.1 类图

当开发面向对象的系统模型时，我们用类图来表示系统中的类和这些类之间的关联。不严格地讲，对象类可以认为是某种系统对象的一般定义。所谓关联就是类与类之间的链接，表示类与类之间具有的某种关系。因此，每一个类都必须包含与它有联系的类的一些信息。

在软件工程过程的前期阶段，当你要开发模型时，我们用对象来表示现实世界的事物，比如病人、处方、医生等。在系统实现阶段，我们总是需要定义额外的实现对象，它们用来提供所需要的系统功能。在这里，强调的是对现实世界中的对象的建模，以反映一部分需求或前期软件设计过程。

UML 中的类图可表示为不同的细节层次。当我们要建立模型时，第一步通常是观察世界，确定重要对象，然后将它们表示成类。最简单的方法是将类名写入方框中，还可以简单地用直线将两个类连接起来表示存在的关联。比如，图 5-8 就是一个简单类图，该类图中有两个类 Patient 和 Patient Record，并且这两个类之间存在关联。

图 5-8 说明了类图的另外一个特征——能够表示出有多少个对象参与这种关联当中。在这个例子中，关联的每一端都标识了一个“1”，意思是这两个类的对象之间是 1 对 1 的关系，也就是说，一个病人只能有一项记录，而一项纪录也只能存储一个病人的信息。在下面例子中会看到，还有许多其他可能的关系，我们可以定义对象的具体数目，也可以像图 5-9 中所示的那样，用“*”表示联系中所涉及的对象数目不确定。

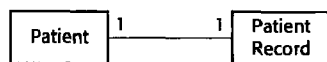


图 5-8 UML 类和关联

图 5-9 所建类图表明类 Patient 的对象还和许多其他类有联系。这个例子表明我们可以对联系命名以便给读者一些类之间关系类型的提示。UML 也允许为参与关联的对象指派角色。

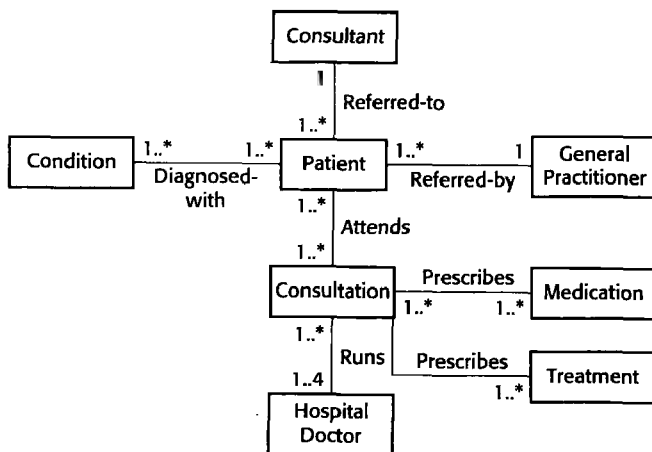


图 5-9 MHC-PMS 中的类和关联

在这个细节层次上的类图看起来像是语义数据模型。语义数据模型在数据库设计时使用，它们表示数据实体、与它们相关的属性，以及实体之间的关系。这种建模的方法是在 20 世纪 70

年代由 Chen (1976) 第一次提出来的。此后发生了几次变化 (Codd, 1979; Hammer 和 McLeod, 1981; Hull 和 King, 1987), 但基本格式都是一样的。

UML 中并没有包括一套专门的符号系统服务于数据库建模, 这是因为它假设所采用的过程是面向对象的开发过程, 并且使用对象以及它们之间的关联为数据建模。然而, 我们可以使用 UML 表示语义数据模型。我们可以把语义数据模型中的实体看做是简化 (不含有操作) 了的对象类, 把属性看做是对象类属性, 把关系看做是对象类间的命名的关联。

在表示类与类之间的关联时, 用尽可能最简单的方法表示这些类可能会给我们带来很大的便利。只要我们增加所定义的类的细节, 就会增加它们的属性 (对象的特性) 和操作 (从对象中可请求的东西) 信息。比如, 一个 Patient 对象如果含有属性 Address, 那么我们就需要添加一个 Change Address 操作, 这在病人变换地址时调用。UML 是通过扩展表示类的简单矩形的方式表示类的属性和操作的。图 5-10 给出了一个示例:

- 1. 对象类的类名位于最上端。
- 2. 类的属性位于中间部分, 其中属性名是必须有的, 属性类型不作强制要求。
- 3. 与对象类中的操作 (在 Java 和其他面向对象编程语言中叫方法) 位于矩形的下部。

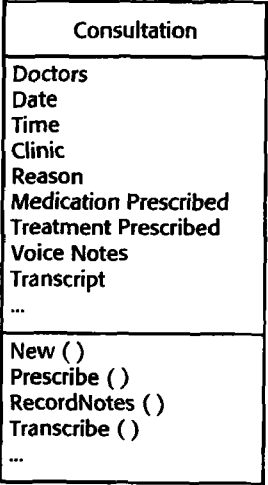


图 5-10 Consultation 类

图 5-10 给出了类 Consultation 可能涉及的属性和操作。在此例中, 假定用录音整理来记录咨询的详细情形。开处方时, 相关医生必须使用开处方方法 Prescribe () 来生成电子处方。

5.3.2 泛化

泛化是一项常用技术, 我们用这种技术来管理复杂性。我们并不是去学习我们所经历的每一个实体的所有细节特性, 而是用更一般的类 (比如, 动物、汽车、房子等) 来表示它们, 然后研究这些类的性质。这样就允许我们推断这些类的成员具有某些共同的特征 (比如, 松鼠和老鼠都是啮齿动物)。我们可以针对类的所有成员给出一般性的描述。

在系统建模中, 检查系统中的类, 看看它们是否还有继续泛化的空间, 这通常是有用的。这意味着共同的信息仅被保持在一个位置。这是很好的设计习惯, 因为当发生变更时, 我们就不必观察系统中的所有类看它们是否受到变更的影响。在面向对象语言中, 比如 Java, 泛化是使用内嵌在语言中的类继承机制来实现的。

UML 中有专门的关联类型来表示泛化, 如图 5-11 中所表示的。泛化用指向上面更一般的类的箭头表示。这表明一般的从业者和住院医生可被泛化为医生, 总共有 3 种类型的住院医生 (Hospital Doctor): 刚从医学院毕业的, 工作中还必须被监督的实习医生 (Trainee Doctor); 工作上不需要监督并且是会诊队伍的主要成员的注册医师 (Registered Doctor); 还有会诊医生 (Consultant), 他们是对所做决定负全责的高级医师。

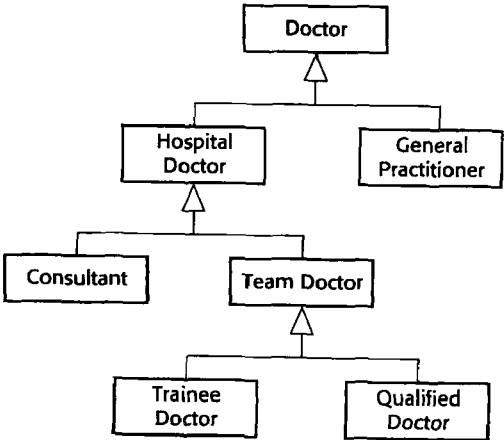


图 5-11 泛化层次结构

在泛化关系中，更高层次上的类中的属性和操作也是较低层次上的类的属性和操作。简言之，低层次的类是子类，它继承了它们的超类的属性和操作。这些低层次类然后添加其他专门属性和操作。例如，所有的医生都有一个名字和电话号码，所有住院医生都有一个工号和部门，但一般从业者不具有这些属性，因为他们是独立工作的。但是他们有一个从业诊所名和地址，如图 5-12 所示。图 5-12 表示的是医生泛化层次的一部分，其中对类扩充了类属性。与医生类相关联的操作是为了注册和注销 MHC-PMS 中的医生。

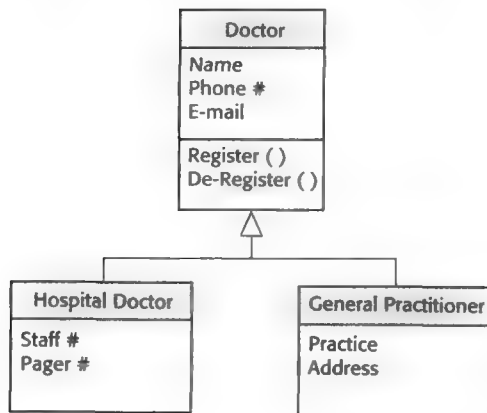


图 5-12 添加了细节的泛化层次结构

5.3.3 聚合

现实世界中的对象通常是由不同部分组成。比如，上课用的学习包可能由书、幻灯片、测试题以及深度阅读建议组成。有时，在系统模型中，我们需要表示这种关系。UML 提供了一种叫做聚合的特殊类型的关联。它意味着一个对象（相当于全体）可以由其他对象（相当于部分）组成。为了表示这种关系，我们用与类相邻的菱形表示全体，如图 5-13 所示。图 5-13 表示一条病人记录由 Patient 和一个不确定的 Consultation 组成。

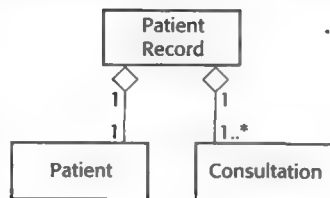


图 5-13 聚合关联

5.4 行为模型

行为模型是描述系统运行时的动态行为的模型，表示当系统响应来自所处环境的刺激时所发生的或有可能发生的事情。这样的刺激有两种：

1. **数据** 一些数据到达必须由系统处理。
2. **事件** 某些触发系统处理的事件发生。事件可能会有相关联的数据，但并不总是这样。



数据流图

数据流图（DFD）是个系统模型，它是从功能角度描述系统，每一个变换代表一个单独的函数或过程。DFD 用来表示数据是如何在一系列处理过程中流动的。比如，一个处理过程可能是过滤客户数据库中重复的记录。数据在经过每一步时被转换一次然后流向下一阶段。这些处理步骤或转换表示软件过程或函数，这里用数据流图来表达软件设计。

<http://www.SoftwareEngineering-9.com/Web/DFDs>

许多业务系统是主要由数据驱动的数据处理系统，它们由输入数据所控制，很少处理外部事件。它们的处理包含一系列在那些数据上的动作以及输出的生成。比如，一电话账单系统将接受用户拨打电话的信息、计算电话费用，并产生发送给用户的账单。相对而言，实时系统主要处理事件而很少涉及数据处理。比如，有线电话切换系统响应如“接收方占线”事件的方法是产生一个拨号音，而响应按键事件的方法是捕获电话号码，等等。

5.4.1 数据驱动的建模

数据驱动模型描述一个动作序列，该动作序列涉及输入数据的处理和相关输出的产生。在需求分析过程中，数据驱动模型非常有用，因此它们可用来表示系统中端到端的过程。也就是说，数据驱动模型表示一个完整的动作序列，从输入处理开始到相关输出的产生，输出可以看成是系统的响应。

数据驱动模型是最早的图形化软件模型之一。在 20 世纪 70 年代，结构化的方法，比如 DeMarco 的结构化分析（DeMarco, 1978），引入了数据流图（DFD）并将其作为一种表示系统中处理步骤的方法。数据流模型有用是因为它跟踪并记录了与一个特别过程相关的数据是如何通过这个系统的，这有助于帮助分析人员和设计人员理解正在发生什么。数据流图不但简单而且直观，另外数据流图还有可能被潜在的系统用户所理解，然后参与到模型验证模型的过程中。

当 DFD 最初被提出来并用于数据过程建模时，UML 不支持数据流图。原因是 DFD 关注的是系统功能而不识别系统对象。然而，因为数据驱动系统在业务中太常用了，所以 UML 2.0 引入了与数据流图类似的活动图。比如，图 5-14 表示了胰岛素泵软件中的处理链。在此图中，我们可以看到处理步骤（用活动表示）和在这些步骤之间流动的数据（用对象表示）。

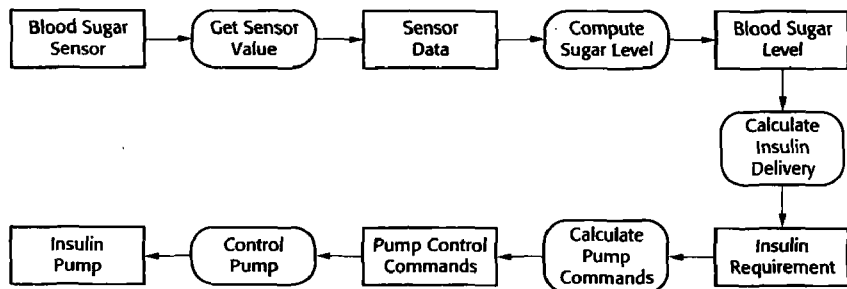


图 5-14 胰岛素泵操作过程

还有一种表示系统处理序列的方法，就是使用 UML 的时序图。我们已经知道了如何用这种图对交互建模，但是，如果我们只让信息从左向右传递，那么它们所表示的就是系统中序列化的数据处理过程，如图 5-15 所示。图 5-15 用了一个时序模型表示对订单的处理并发送给供应商。顺序模型强调系统中的对象，而数据流图关注的是功能。订单处理的等价的数据流图在本书网页上有图示。

5.4.2 事件驱动模型

事件驱动模型表示系统对内外部事件的响应方式。下面讨论是以一个假设为基础的，即系统状态是有限的，并且事件（激励）可能引起从一种状态向另一种状态的转变。比如，当控制阀门的系统接收到操作员的指令（激励）时，可能从状态“阀门开”变为状态“阀门闭”。这种

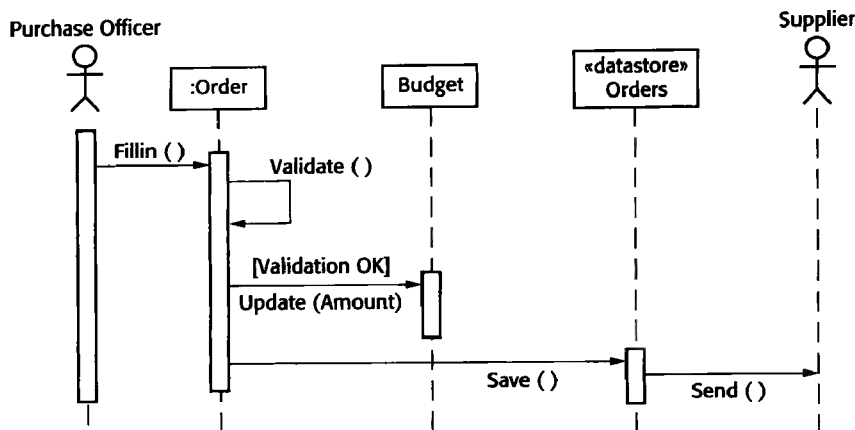


图 5-15 订单处理过程

系统观点特别适合于实时系统。基于事件的模型方法被引入到实时设计方法中，比如 Ward 和 Mellor (1985) 以及 Harel (1987, 1988) 所提议的方法。

UML 通过使用状态图支持基于事件的模型，所用状态图以 Harel 所提出的 Statecharts (Harel, 1987, 1988) 为基础变化而来的。状态图表示系统状态和引起状态改变的事件。状态图不表示系统中的数据流，但可能包括在每一状态中所执行运算的附加信息。

下面使用一个非常简易的微波炉的控制软件来阐述一下事件驱动建模，真实的微波炉通常比这个系统要复杂得多，但为了容易理解，我们将系统简化了。这个简单的微波炉有一个用来选择全功率和半功率的开关，一个供我们输入烹饪时间的数字键盘，一个开始/停止按钮，一个能显示字母和数字的显示器。

我们假定使用微波炉的动作顺序为：

1. 选择功率水平（半功率或全功率）。
2. 用数字键盘输入烹饪时间。
3. 按下开始按钮，烹饪食物到指定的时间。

出于安全原因，微波炉在没关炉门时不能工作，并且烹饪一完成，就要响起蜂鸣声。微波炉有一个非常简单的能显示字母和数字的显示器，该显示器被用来显示各种警报和警告信息。

在 UML 状态图中，圆边矩形代表系统状态，其中可能包括此状态中所执行动作的简单描述（以“Do”引出）。带标签的箭头代表促使系统从一种状态变为另一种状态的激励因素。如在活动图中一样，用实心圆表示状态开始和状态结束。

从图 5-16 可以看出，系统开始时处于等待状态，全功率或半功率按钮都可使系统发生响应，当使用者选择其中一个按钮后可以改变想法按下另外一个按钮。设置好时间并且关上炉门，就可以按下开始按钮，然后微波炉就开始工作一直到设定的时间。等到一个烹饪周期完成，系统回到等待状态。

我们使用 UML 符号来表示在一个状态中所发生的活动。在详细的系统描述中，我们必须提供更多有关激励因素和系统状态方面的细节。图 5-17 以表格的形式描述了微波炉系统的每一个状态和促使系统状态发生改变的激励因素。

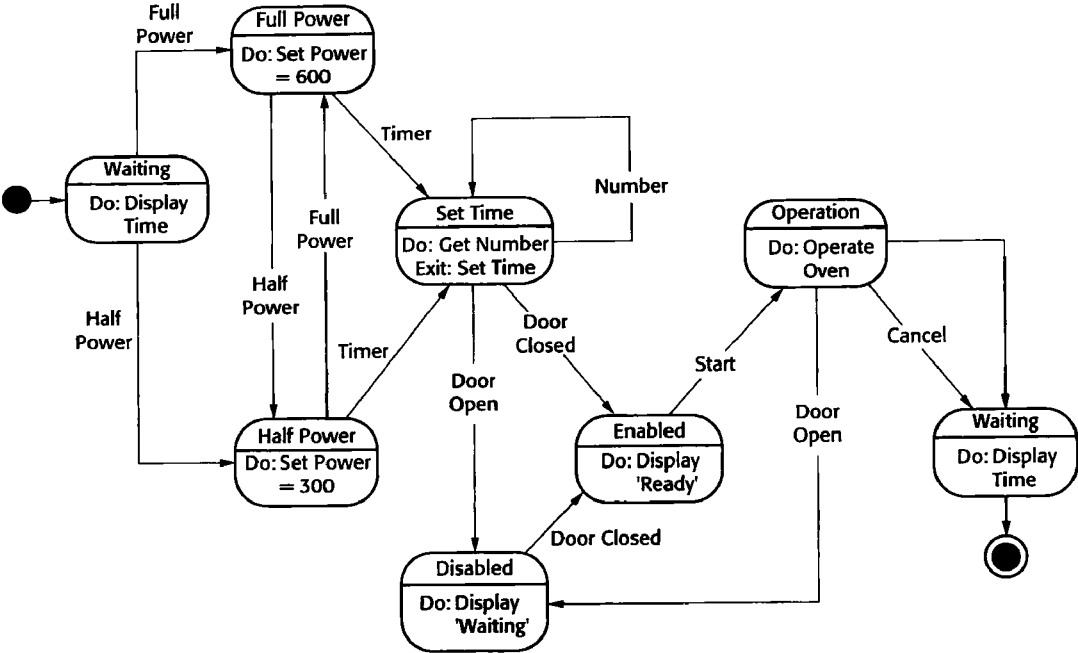


图 5-16 微波炉的状态图

状 态	描 述
等待	微波炉等待输入，显示器显示当前时间
半功率	微波炉的功率设置为 300W，显示为“半功率”
全功率	微波炉的功率设置为 600W，显示为“全功率”
设置时间	烹饪时间设置为用户输入值。显示器显示所选择的烹饪时间，当更改输入时更新显示
屏蔽	为了安全起见，微波炉操作被屏蔽。炉内灯光点亮
激活	微波炉操作重新可用，炉内灯光熄灭。显示器显示“准备就绪”
运行	微波炉工作。炉内灯点亮。显示器显示逐渐减少的时间。当烹饪完成，蜂鸣器将响 5 秒。炉内灯持续亮着。在蜂鸣器响的同时显示器显示“烹饪完成”
激 励	描 述
半功率	用户按下半功率按钮
全功率	用户按下全功率按钮
定时器	用户按下某个定时器按钮
数字	用户按下某个数字键
门开启	微波炉门开关没有关闭
门关闭	微波炉门开关闭
启动	用户按下了启动按钮
取消	用户按下了取消按钮

图 5-17 微波炉的状态和激励

以状态为基础的建模遇到的问题是状态的数量可能急剧增加。因此，对于大的系统模型，我

们需要在模型中隐藏一些细节。其中一种方法就是使用超态（superstate）的概念，超态是封装了多个独立的状态而得到的一种状态。这种超态看起来像一个高层模型中单独的一个状态，但是之后会在一个独立图中被扩展来表示更多的细节。为了表述这个概念，图 5-18 给出了图 5-16 中操作状态的一个可扩展的超态。

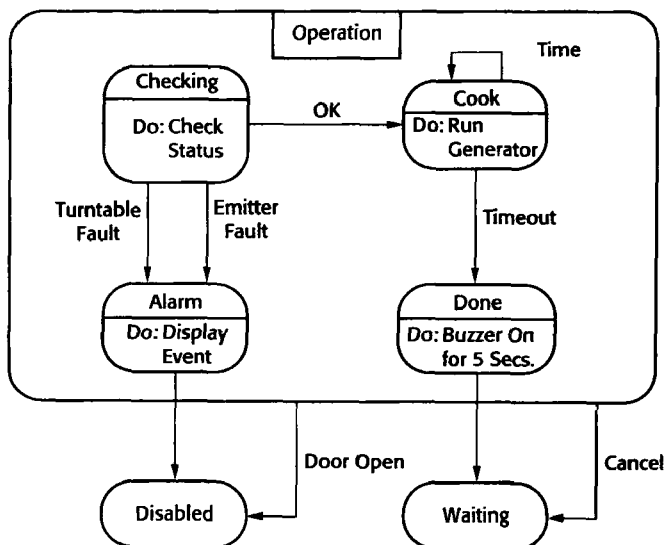


图 5-18 微波炉操作

运行状态包括许多子状态。图中表明操作开始首先进行状态检测，当检测到任何问题时都会有警告显示，并且操作失败。烹饪过程即让微波产生器运行所指定的时间，烹饪一旦完成，就会响起蜂鸣声。正如图 5-16 所示的，如果在操作过程中，炉门被打开，则系统变为不可操作状态。

5.5 模型驱动工程

模型驱动工程（MDE）是软件开发的一种方法，在这种方法中，模型而不是程序成为开发过程中的主要输出（Kent, 2002; Schmidt, 2006）。在硬件/软件平台上执行的程序会由模型自动生成。MDE 的支持者认为这样做提高了软件工程的抽象水平，使得工程师不必再去关心编程语言的细节或执行平台的特性。

模型驱动工程起源于模型驱动体系结构，这是在 2001 年由对象管理集团（OMG）提出来的，并被作为一种新的软件开发范例。模型驱动工程和模型驱动体系结构通常被认为是同一个东西，然而，MDE 涉及面比 MDA 要宽一些。本节稍后会讲到，MDA 关注的是软件开发中的设计和实现阶段，而 MDE 关心软件工程的所有方面。因此，像以模型为基础的需求工程，软件开发过程以及测试问题都是 MDE 所涉及的问题，而目前还没有用 MDA 来解决这些问题。

尽管 MDA 从 2001 年就已经开始使用，但以模型为基础的工程仍停留在开发的早期阶段，并且它对软件工程的开发是否会产生重要影响也是不明确的。下面是人们对 MDE 的两种不同看法：

1. **支持者** 认为以模型为基础的工程允许工程师在更高的抽象水平上考虑系统，而不用关心它们实现的细节。这样减少了出错的可能性，加速了设计和实现过程，并且支持可复用的、具有独立平台的应用模型的建立。通过使用强有力的工具，同一模型所表示的系统可以在不同平台上被实现。因此，为了让系统适应一些新的平台技术，只需要为那个平台写一个翻译器。当这些实现时，所有具有独立平台的模型都可以迅速地在新平台上实现。

2. 反对者 认为,模型是一种很好的便于讨论软件设计的方法。然而,这并不是说模型所支持的抽象都是正确的、能够实现的抽象。所以,我们可能建立非形式化的设计模型,但在其后却使用商业现货软件实现系统。另外,对于平台独立性的讨论,只有对大的长生命周期的系统才成立。这时平台在系统生存期中逐渐被抛弃。然而,对于这类系统,我们知道其主要的问题不是实现——需求工程、信息安全性和可依赖性、与遗留系统的集成以及测试,这些更为重要。

在 OMG 网页 (www.omg.org/mda/products_success.htm) 上有很多 MDE 的成功使用案例,并且这种方法在像 IBM 和西门子这样的大公司中也有使用。这种技术已经在航空管制系统这样长周期的软件系统开发过程中得到成功应用。然而,到目前为止,模型驱动方法还没有广泛用于软件工程。像将在第 12 章中讲述的形式化方法之于软件工程,MDE 将是一种重要的开发方法,然而,也正是如形式化方法的情形,我们现在不清楚模型驱动方法的成本和风险是否会超过它给我们带来的便利。

5.5.1 模型驱动体系结构

模型驱动体系结构 (Kleppe 等, 2003; Mellor 等, 2004; Stahl 和 Voelter, 2006) 是一种以模型为中心进行软件设计和实现的方法,它使用 UML 模型的子集来描述系统。在这里需要建立不同抽象层次的模型。原则上,对于一个高水平的平台独立的模型来讲,在没有人为干预情况下产生一个工作程序是有可能的。

MDA 方法认为应该产生 3 种类型的抽象系统模型。

1. 计算独立模型 (CIM), 此模型为系统中使用的重要的领域抽象建模。CIM 有时也被称为领域模型,我们可能建立几个不同的 CIM 来反映系统的不同方面。比如,安全 CIM,其中定义了比如资产、角色等一些重要的安全抽象概念;病人信息记录 CIM,其描述的是病人、会诊等一些抽象概念。

2. 平台独立模型 (PIM), 此模型在没有它的实现作为参考下为系统的运行建模。我们通常使用 UML 模型来描述 PIM, UML 模型表示静态系统结构和系统对内外部事件的响应。

3. 平台特定模型 (PSM)。PSM 是由平台独立模型转化得到的,其中每一个应用平台都有一个独立的 PSM。原则上,PSM 应有好几层,每一层都增加了一些特定的平台细节。所以,第一层 PSM 可能是中间件,专门的但独立于数据库的。当选定一个特定的数据库时,一个数据库专门的 PSM 就产生了。

前面已经讲到,模型之间的转换由软件工具自动完成,如图 5-19 所示。该图也表示了自动转换的最后一层。PSM 可通过变换生成在所设计的软件平台上运行的可执行代码。

现在为止,CIM 向 PIM 的自动翻译仍处在研究原型阶段。在近期还不可能出现完全的自动翻译工具。人工干预,在图 5-19 中用小人表示,在可预期的未来还是需要的。CIM 之间是有联

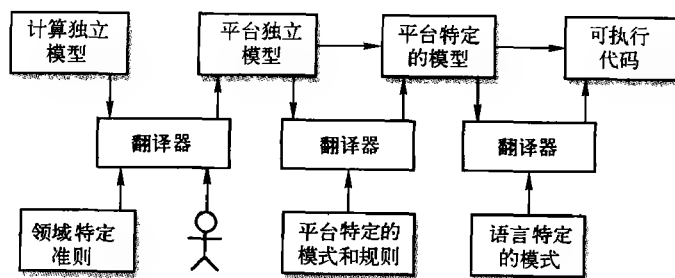


图 5-19 MDA 变换

系的并且一部分翻译过程可能涉及不同 CIM 中的概念的链接。比如，安全 CIM 中角色的概念可能映射到医院 CIM 中的工号的概念。Mellor 和 Balcer (2002) 给出了“桥”的概念来描述支持 CIM 之间映射的信息。

从 PIM 向 PSM 的翻译更加成熟，并且已有一些商业工具提供从 PIM 向通用平台比如 Java 和 J2EE 的翻译器。它们依靠一个很大的平台特定的规则和模式库将 PIM 转换为 PSM。系统中有可能多个 PSM 对应一个 PIM。如果一个软件系统要在不同平台上运行（比如 J2EE 和 .Net），那么唯一必须的就是维护 PIM。每一个平台的 PSM 是自动产生的，如图 5-20 所示。

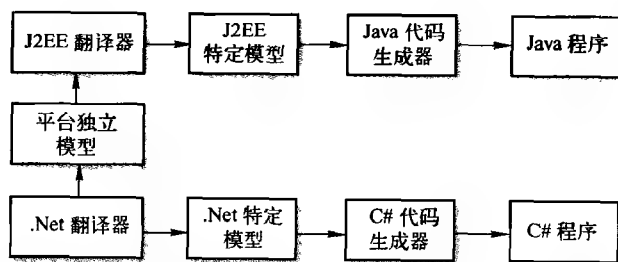


图 5-20 复合平台特定模型

尽管支持 MDA 的工具包括平台特定的翻译器，一般情形是它们只提供从 PIM 向 PSM 翻译的部分支持。大多数情况下，一个系统的执行环境往往不止是标准的执行平台（比如 J2EE 和 .Net）。还包括其他的应用系统、公司专用的应用库，以及用户界面库。因为各公司间的差异很大，所以标准的工具支持就失效了。因此，当介绍 MDA 时，必须建立具有特殊目的翻译器，它们应将本地环境的特点考虑进去。有些情况下（比如用户界面的产生），PIM 完全自动翻译为 PSM 几乎是不可能的。

敏捷方法和模型驱动体系结构之间的关系不是很顺畅。大规模的前期建模和敏捷宣言中的基本思想有冲突，很少会有敏捷开发者会对模型驱动工程感到很舒服。MDA 的开发者认为，MDA 支持迭代的开发方法所以可以应用在敏捷方法中（Mellor 等，2004）。如果 PIM 可进行完全自动的转换并产生完整的程序，那么，原则上，MDA 就可以应用在敏捷开发过程，因为不再需要单独编写代码。然而，据作者所知，现在还没有能支持回归测试和测试驱动开发活动的 MDA 工具。

5.5.2 可执行 UML

模型驱动工程背后的基本的概念是，模型向代码的完全自动化转化是可行的。为了实现这一点，我们必须能够建立语义完好定义的图模型。我们还需要一种向图模型添加信息的方法，以保证模型中的定义的操作能够被实现。这是可行的，通过使用 UML 2.0 的子集，即可执行 UML 或 xUML（Mellor 和 Balcer，2002）。这里没有足够的篇幅介绍 xUML 的具体内容，仅简单地给出一个对 xUML 的主要特征的介绍。

UML 是一种支持软件设计和记录软件设计的语言，而不是编程语言。UML 的设计者关心的不是语言的语义细节，而是它的表达能力。他们引入了一些有用的概念，比如用例图，以帮助设计，但是它们是很不形式化的所以不能支持执行。为了建立一个 UML 可执行的子集，模型类型的数量已经被剧减为 3 个主要的模型类型：

1. 领域模型，识别出系统的主要关注点，然后由 UML 的类图定义，其中包括对象、属性和关联。
2. 类模型，在类模型中定义类以及它们的属性和操作。

3. 状态模型，状态模型中的每一个状态图都与一个类相关联，并且用来描述类的生命周期。

系统的动态行为可以使用对象约束语言（OCL）详细说明，或者使用 UML 的动作语言表达。这种动作语言就像一种非常高级的编程语言，我们可以用它来查找对象和它们的属性，还可指定将执行的动作。

要点

- 模型是忽略了一些系统细节的系统抽象表示。建立互补的系统模型来表示系统的上下文、交互、结构和行为。
- 上下文模型描述所建模的系统是如何在含有其他系统和流程的环境中工作的。它们帮助定义被建系统的边界。
- 用例图和时序图用来描述系统用户之间或系统用户和其他系统之间的交互。用例描述的是系统和外部参与者之间的交互；时序图通过表示系统对象之间的交互为用例图添加更多的信息。
- 结构模型表示的是系统的体系结构。类图用来定义系统中类的静态结构以及类之间的关联关系。
- 行为模型用来描述可执行系统的动态行为。我们可以从系统处理数据的角度和事件激励系统产生响应的角度来建立这种模型。
- 活动图用来为数据的处理过程建模，其中每一个活动图代表一个处理步骤。
- 状态图用来为响应内外部事件的系统行为建模。
- 模型驱动工程是一种软件开发的方法，其中的系统表示为可自动转换为可执行代码的模型的集合。

进一步阅读材料

《Requirements Analysis and System Design》这本书主要关注信息系统分析，讨论如何在分析过程中使用不同 UML 模型（L. Maciaszek, Addison-Wesley, 2001）。

《MDA Distilled: Principles of Model-driven Architecture》它是对 MDA 方法的一个通俗易懂的介绍。因为这本书由支持者所著，所以这本书对这种方法可能存在的问题讲得很少（S. J. Mellor, K. Scott and D. Weise, Addison-Wesley, 2004）。

《Using UML: Software Engineering with Objects and Components, 2nd ed》它简明介绍了 UML 在系统描述和设计中的使用。尽管这本书没有给出 UML 中所有的符号描述，但它绝对是学习和理解 UML 的一本好书（P. Stevens with R. Pooley, Addison-Wesley, 2006）。

练习

- 5.1 解释为正在开发的系统的上下文建立模型的重要性，并给出两个由于软件工程师不理解系统的上下文而可能产生的错误。
- 5.2 如何使用现有系统的模型？解释这样的系统模型必须保证完整和正确的原因。如果我们正在建立一个新系统的模型，这样的模型还行吗？
- 5.3 请你开发一个系统，该系统用来帮助我们组织大规模的活动和聚会，比如，婚礼，毕业典礼，生日聚会等。使用 UML 活动图为系统过程上下文建立模型，来表示组织聚会中的活动（比如，预定地点，组织邀请等）和在每一阶段所使用的系统元素。
- 5.4 对 MHC-PMS，请提出一组用例，来表示医生（查看病人并为病人提供治疗、开处方的人）与 MHC-PMS 之间的交互。

- 5.5 请建立一个时序图表示大学生选课时所涉及的交互。因为课程选择是有限制人数的，所以选课过程必须包括对空间有效性的检测。假定学生可以使用一个电子课程目录来发现可选课程。
- 5.6 仔细观察你所使用的电子邮件系统中的信件和电子信箱是怎么表示的。为在系统实现中可能用到的对象类建模来表示电子信箱和电子邮件。
- 5.7 基于你使用银行 ATM 机的经历，请画一个活动图，当客户从机器中提取现金时，为可能涉及的数据处理过程建模。
- 5.8 为同一个系统画一个时序图。解释为什么你在为系统行为建模时，可能会同时建立活动图和时序图。
- 5.9 画出下列控制软件的状态图：
 - 对不同类型的衣服具有不同程序的自动洗衣机；
 - DVD 播放器的软件；
 - 一个电话应答系统。它可以记录发进来的信息并且可以在 LED 上显示接收信息的号码。这个系统应该允许电话客户从任何地点打进来，可以为一系列的号码分类，并可查看任何被记录的信息。
- 5.10 假如你是一个软件工程管理者的，你的团队提出用模型驱动工程开发一个新系统。当决定是否使用这种软件开发的新方法时，你应该考虑哪些因素？

参考书目

- Ambler, S. W. and Jeffries, R. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons.
- Booch, G., Rumbaugh, J. and Jacobson, I. (2005). *The Unified Modeling Language User Guide, 2nd ed.* Boston: Addison-Wesley.
- Chen, P. (1976). 'The entity relationship model—Towards a unified view of data'. *ACM Trans. on Database Systems*, 1 (1), 9–36.
- Codd, E. F. (1979). 'Extending the database relational model to capture more meaning'. *ACM Trans. on Database Systems*, 4 (4), 397–434.
- DeMarco, T. (1978). *Structured Analysis and System Specification*. New York: Yourdon Press.
- Erickson, J. and Siau, K. (2007). 'Theoretical and practical complexity of modeling methods'. *Comm. ACM*, 50 (8), 46–51.
- Hammer, M. and McLeod, D. (1981). 'Database descriptions with SDM: A semantic database model'. *ACM Trans. on Database Sys.*, 6 (3), 351–86.
- Harel, D. (1987). 'Statecharts: A visual formalism for complex systems'. *Sci. Comput. Programming*, 8 (3), 231–74.
- Harel, D. (1988). 'On visual formalisms'. *Comm. ACM*, 31 (5), 514–30.
- Hull, R. and King, R. (1987). 'Semantic database modeling: Survey, applications and research issues'. *ACM Computing Surveys*, 19 (3), 201–60.
- Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. (1993). *Object-Oriented Software Engineering*. Wokingham.: Addison-Wesley.
- Kent, S. (2002). 'Model-driven engineering'. *Proc. 3rd Int. Conf. on Integrated Formal Methods*, 286–98.
- Kleppe, A., Warmer, J. and Bast, W. (2003). *MDA Explained: The Model Driven Architecture—Practice and Promise*. Boston: Addison-Wesley.

Kruchten, P. (1995). 'The 4 + 1 view model of architecture'. *IEEE Software*, **11** (6), 42–50.

Mellor, S. J. and Balcer, M. J. (2002). *Executable UML*. Boston: Addison-Wesley.

Mellor, S. J., Scott, K. and Weise, D. (2004). *MDA Distilled: Principles of Model-driven Architecture*. Boston: Addison-Wesley.

Rumbaugh, J., Jacobson, I. and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Reading, Mass.: Addison-Wesley.

Rumbaugh, J., Jacobson, I. and Booch, G. (2004). *The Unified Modeling Language Reference Manual, 2nd ed.* Boston: Addison-Wesley.

Schmidt, D. C. (2006). 'Model-Driven Engineering'. *IEEE Computer*, **39** (2), 25–31.

Stahl, T. and Voelter, M. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. New York: John Wiley & Sons.

Ward, P. and Mellor, S. (1985). *Structured Development for Real-time Systems*. Englewood Cliffs, NJ: Prentice Hall.

体系结构设计

目标

本章的目标是介绍软件体系结构概念和体系结构的设计。读完本章，你将了解以下内容：

- 软件的体系结构设计为什么很重要；
- 在体系结构设计阶段对系统体系结构必须做出的决策；
- 介绍体系结构的模式思想，介绍经验证明有效的、并且能够在系统设计中复用的组织系统体系结构的方法；
- 了解经常在不同的应用系统类型中使用的体系结构模式，包括事务处理系统和语言处理系统。

体系结构设计关心的是理解如何组织一个系统和设计系统的整体结构。如在第2章所介绍的，在软件开发过程模型中，体系结构设计是软件设计过程中的第一步。体系结构设计是设计和需求工程之间的桥梁，因为它识别系统的主要的结构组件以及它们之间的关系。体系结构设计过程的输出是体系结构模型，它描述系统是如何由一组相互交互的组件组成的。

在敏捷过程中，人们普遍接受这样一种观念，即开发过程的早期应当涉及建立整体的系统体系结构。体系结构的增量式开发并不总是成功的。虽然以重构组件来响应变更通常是相对容易的，但是重构一个系统体系结构可能是非常昂贵的。

为了帮助读者理解系统体系结构，考虑图6-1。该图显示了一个打包机器人系统的体系结构的抽象模型，它描述了所要开发的子系统。这个机器人系统能够对不同类型的对象进行打包。它使用了一个视觉子系统来拾取传送带上的对象，识别对象类型并选择正确的打包方式，然后从传送带上移下对象、打包，最后将其送到另一个传送带上。体系结构模型描述了这些组件以及它们之间的关联。

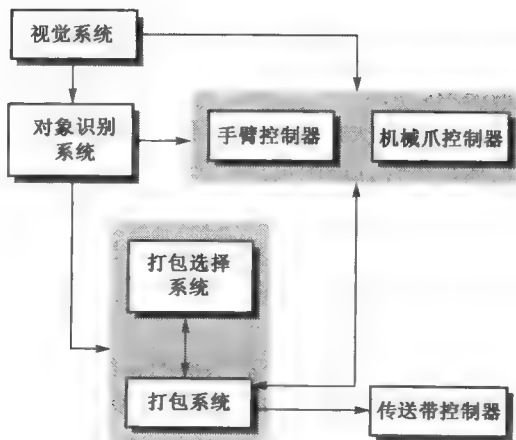


图 6-1 打包机器人控制系统的体系结构

在实际过程中，在需求工程和体系结构设计之间有很大的重叠。理想情形是，系统描述应该

不包含设计信息。但这是不现实的,除非对于很小的系统。体系结构分解对于结构和组织描述是必要的。因此,作为需求工程过程的一部分,我们会提供一个抽象的系统体系结构,把许多系统功能和特征集合关联到一些大规模组件或者子系统上。那么我们就能够使用这种分解来与信息持有者讨论需求和系统特征。

我们可以在两个抽象层次上设计软件体系结构,它们是小规模体系结构和大规模体系结构:

1. 小规模体系结构与单个程序的体系结构相关。在这个层次上,我们关注将单个程序分解成许多组成部分的方法。这一章主要考虑程序体系结构。

2. 大规模体系结构与复杂企业系统的体系结构相关,包括其他的系统、程序和程序组件。这些企业系统分布在不同的计算机上,这些计算机可能由不同的企业拥有和管理。第18章和第19章涉及大规模体系结构,在那里我们将讨论分布式体系结构。

软件体系结构非常重要,因为它影响系统的性能、健壮性、分布能力和可维护性(Bosch, 2000)。就如Bosch所述,单个的组件实现系统功能需求,而非功能需求依靠体系结构——一种把这些组件组织起来并能够通信的方法。在许多系统中,非功能需求也会受到单个组件的影响,但是毫无疑问系统的体系结构起到决定性的作用。

Bass等(2003)讨论了明确设计和文档化软件体系结构的3个好处:

1. **信息持有者之间的沟通** 体系结构是系统的一个高层表示,可以作为不同的信息持有者之间讨论的焦点。

2. **系统分析** 在系统开发的早期阶段明确给出系统的体系结构,实际上就是对系统的分析过程。体系结构设计决策对系统能否满足关键性需求(如系统的性能、可靠性和可维护性等)具有极深的影响。

3. **大规模复用** 系统体系结构的模型是一个紧凑的、易于管理的描述,描述系统如何组织和组件间如何交互。体系结构能在具有相似需求的系统之间互用,由此来支持大规模的软件复用。如第16章中所讨论的,我们可以开发出产品线体系结构,来供一系列相关系统使用。

Hofmeister等(2000)提出软件体系结构首先可以作为一个设计计划,用来协商系统需求,其次可以作为与客户、开发人员以及管理人员进行结构讨论的手段。他们还建议,这是复杂性管理的一个重要工具。它隐藏了细节,并允许设计者集中于一些关键性的系统抽象。

系统体系结构经常会用方块图来建模,如图6-1所示。在方块图中每一个方块代表一个组件,方块中的方块表示所代表的组件又被分解为一些子组件,箭头表示所示的方向有数据和控制信号从一个组件流动到另一个组件。在Booch的软件体系结构目录中(Booch, 2009)我们能看到许多这种类型的体系结构模型的例子。

方块图给出了系统结构的一个高层次描述,来自不同工程学科的、参与到此系统开发过程中的人员都能够看得懂这种描述。尽管方块图被广泛地应用,但Bass等(2003)并不喜欢以非形式化的方块图来描述体系结构。他们认为这种非形式化的方块图并不是一种好的体系结构的表示方法,因为它既没有给出系统组件之间的关系类型,也没有显示出组件可见的外部特性。

在实践和体系结构理论之间出现了很明显的矛盾,因为使用程序的体系结构模型有两种方法:

1. **作为一种加快系统设计商讨的方法** 一个高层次的系统体系结构视图对于与信息持有者之间沟通以及项目规划是非常有效的,因为它不掺杂细节。信息持有者能看得懂,了解系统的概貌。他们能够把系统看作一个整体来讨论而不被细节所混淆。体系结构模型能识别出每个将要开发的关键组件,所以管理者可以指派人员规划系统的开发。

2. **作为一种对已设计好的体系结构文档化的方法** 这里的目标是制作出一个完全的系统模型,来显示系统中的不同组件以及它们之间的接口和关联。这种方法的理由是,如此详细的体系

结构描述使得对系统的理解和进化变得容易。

在设计过程中，方块图是描述系统体系结构很合适的方法，因为它是一种支持设计过程中的人与人之间沟通的很好的方式。在许多项目中，方块图通常是仅存的体系结构文档。可是，如果完全地记录系统体系结构，那么最好用一种带有清晰语义的符号系统来描述体系结构。尽管如此，就像将在 6.2 节中提到的，许多人认为详细的文档既没用而且开发它的花费也确实是十分昂贵的。

6.1 体系结构设计决策

体系结构设计是一个充满创造性的过程，设计一个系统组成来满足功能性和非功能性系统需求。由于它是一个创造性的过程，所以工程中的活动很大程度上依赖于所要开发的系统类型、系统架构师的背景和经验以及系统的特殊需求。因此，把体系结构设计看做一系列将要做的决策要比把它视为一连串的活动更有效。

在体系结构设计过程中，系统架构师必须做出很多结构上的决定，这些决定将极大地影响系统本身及其开发过程。根据他们的知识和经验，他们必须回答以下一些关于系统的根本性问题：

1. 对于所要设计的系统，是否存在一个一般性的应用体系结构可以当做模板？
2. 系统将如何分配到多个核或处理器上？
3. 哪些体系结构模式和风格可能会用到？
4. 哪种方法是系统结构分析的基本方法？
5. 如何将系统中的结构组件分解为子组件？
6. 应该使用什么样的策略来控制系统中组件的操作？
7. 什么样的体系结构组织方式对于交付非功能性的系统需求是最好的？
8. 如何评估体系结构设计？
9. 如何对系统体系结构文档化？

尽管每个软件系统都是独特的，但是同一个应用领域内的系统通常具有相似的体系结构，这种体系结构能反映基本的领域概念。例如，应用产品线是围绕一个核心体系结构建立的一些应用，核心体系结构会有一些变量，可以调整这些变量来满足专门用户的需求。在设计系统体系结构时，我们必须找出本系统和更广泛的一些应用类之间的共同之处，然后确定从这些应用体系结构中能够复用多少。我们将在 6.4 节中讨论一般应用体系结构，在第 16 章中讨论应用产品线。

对于嵌入式系统和为个人计算机设计的系统，通常只有一个处理器，我们无需设计分布式体系结构。然而，绝大多数大型系统现在都是分布式系统，系统软件是分布在很多不同计算机上的。对分布式体系结构的选择是一个关系到系统性能和可靠性的关键决断。这本身就是一个重要话题，将在第 18 章中单独介绍。

软件系统的体系结构要基于特定的体系结构模式或风格。体系结构模式是系统组成的描述 (Garlan 和 Shaw, 1993)，比如客户机 - 服务器组成结构和分层体系结构。体系结构模式抓住了体系结构的本质，这种本质的东西已经在不同的软件系统中得到运用。当要对系统体系结构做出决策的时候，我们应当了解通用模式，它们都适合在哪里使用，以及它们各自的优点和缺陷。6.3 节将讨论一些经常用到的模式。

Garlan 和 Shaw 关于体系结构风格的概念（风格和模式是同一个意思）涉及前面根本性问题列表中的 4 到 6 号问题。我们必须选择最合适的结构，比如客户机 - 服务器结构或者是分层结构，这能让我们完成系统需求。为了分解结构性系统单元，我们得制定将组件分解为相应的一些

子组件的策略。所能使用的方法允许我们去实现不同类型的体系结构。最后，在控制建模过程中，做出关于如何控制组件的执行的决策。对系统的不同部分之间的控制关系，要开发一个通用模型。

由于非功能性需求和软件体系结构的密切关系，我们为系统所选择的特殊的体系结构风格和结构应当依赖于非功能性系统需求：

1. **性能** 如果性能是关键性需求，那么体系结构的设计就要定位在少数几个组件上的一些关键操作上，把这些组件都部署到同一台计算机而不是分配到网络中。这可能意味着系统要使用较大粒度的组件而不是小粒度的组件，以此来减少组件之间的通信次数。我们可能还会考虑运行时系统组织方式，允许系统在不同的处理器上被复制和执行。

2. **信息安全性 (security)** 如果信息安全性是一个关键性的需求，那么体系结构的设计就要采用分层结构，把重要资源放在内层，并且在每层中采用更加严格的信息安全有效性验证。

3. **安全性 (safety)** 如果安全是一个关键性的需求，那么系统体系结构的设计就要将与安全相关的操作集中在一个或少数几个组件中。这样做会降低成本和减少安全有效性验证方面的问题，而且有可能为之提供相应的保护系统，这种保护系统能够安全地关闭系统以防失败。

4. **可用性** 如果可用性是一个关键性需求，那么系统体系结构设计就要采用冗余组件以便能在无需系统停止运行的情况下更新和替换组件。第 13 章描述了两种适于高可用性系统的容错型体系结构。

5. **可维护性** 如果可维护性是一个关键性需求，那么系统体系结构设计就要使用小粒度的自包含组件以便于更换。数据的生产者和数据的消费者应该分开，尽量避免它们之间的数据共享。

显然这些体系结构之间有潜在的冲突。举例来说，性能的改进需要使用大粒度组件，而改善可维护性需要使用小粒度组件。如果这两个指标都是系统的关键性需求，就需要采取一个折中方案。有时可以通过针对不同部分而采用不同的体系结构模式或风格来实现。

对体系结构设计的评估是困难的，这是因为，对体系结构的真正的检验是要看当它正在使用的时候能在多大程度上满足功能性和非功能性需求。然而，在某些情况下，我们可以通过将自己的设计和参考体系结构或一般体系结构模式进行比较来评估。Bosch (2000) 对于体系结构模式非功能性特征的描述也能用来帮助评估体系结构。

6.2 体系结构视图

在本章的引言中解释了软件体系结构模型能够用来聚焦关于软件的需求和设计的讨论。或者，它可以用来文档化设计过程以便作为更详细设计和实现的基础，并且有利于系统未来的进化。本节讨论以下两个相关的问题：

1. 当设计和文档化一个体系的体系结构时，什么样的视图或角度是有效的？
2. 应该使用什么样的符号系统来描述体系结构模型？

在单个的体系结构模型中是不可能提出所有与系统体系结构相关的信息的，因为每一种模型只能显示系统的一种角度和视图。它能说明系统是如何分解成许多的模块，运行时进程之间是如何交互的，或系统组件通过网络是如何以不同的方式分布的。对于设计和文档化，在不同的时期，所有这些都是非常有用的，我们通常需要提供系统体系结构的多重视图。

至于需要什么样的视图是有很多选择的。Krutchen (1995) 在其众所周知的系统体系结构 4+1 视图模型中建议应该有 4 种基础的体系结构视图，这些视图通过用例或脚本关联在一起。他建议的视图有：

1. **逻辑视图**，它显示了系统中对象和对象类的一些主要抽象。通过这种逻辑视图，将系统

需求和实体关联起来应该是可能的。

2. 进程视图，它显示了在运行时系统是如何组织为一组交互的进程。这种视图对非功能系统特征的判断是非常有效的，比如性能和可用性。

3. 开发视图，它显示了软件是如何为了开发而被分解的，即将软件分解成可以由单独的开发人员或开发团队实现的组件。这种视图对软件的管理者和程序员有用。

4. 物理视图，它显示了系统硬件和系统中软件组件是如何分布在处理器上的。这种视图对系统工程师规划系统部署是非常有用的。

Hofmeister 等（2000）建议在使用类似视图的基础上再添加概念视图。这种视图是系统的抽象视图，它可以作为把高层次需求分解为详细描述的基础，来帮助工程师在可复用的组件、表现产品线而不是单独的系统（在第16章中讨论）等方面做出决策。如图6-1所示，描述了打包机器人的体系结构就是概念性系统视图的一个例子。

实际上，在设计过程中几乎常常会形成概念视图，同时它对体系结构的决策也是很有帮助的。概念视图给出系统的本质内容供不同的信息持有者之间交流。在设计过程中，当讨论系统的不同方面时也可能会形成一些其他的视图，但是包含各个角度的完全描述是没有必要的。关于系统的不同视图也可能会涉及体系结构模式，第7章会讨论这一主题。

关于软件体系结构是否应该使用UML来描述有不同的看法（Clements等，2002）。2006年的一个调查显示（Lange等，2006），UML被使用的情形，多数都是松散的、非形式化的场合。这篇论文的作者认为这是一件不好的事情。本书作者却不同意这种观点，设计UML是为了描述面向对象系统，在体系结构设计阶段，我们常常想要以更高层次的抽象化来描述系统。对象类太接近实现，对于体系结构的描述意义不大。

就设计过程本身而言，看不出UML很有用，推荐采用一些非正式的标记，这些标记写起来更快而且容易在白板上画出来。当我们详细地文档化一个体系结构或使用模型驱动开发的时候，UML是非常有价值的。

有许多研究人员提出了使用专门的体系结构描述语言（ADL）来描述系统体系结构（Bass等，2003）。ADL语言的基本要素是组件和连接器，这类语言包含了形成规范化体系结构所应该使用的规则和指南。不过，由于它的专业性，领域和应用专家很难理解和使用ADL，这导致在实际的软件工程中很难看到它的有效性。ADL语言是为特定领域设计的（例如，汽车系统），或许可以作为模型驱动开发的基础。尽管如此，像UML这种非正式的模型和符号系统将仍然是文档化系统体系结构最普遍使用的方法。

敏捷方法的使用者认为详细的设计文档通常用不到，因此，开发它是在浪费时间和金钱。作者很赞同这种观点并且认为从这4个方面来看，大多数系统是不值得开发一个详细的体系结构描述的。我们应当开发出这种视图，它有益于沟通而且不在乎我们的体系结构文档是否完整。不过，一种例外的情况是，当我们正在开发关键性系统，当我们需要做一个详细的系统可依赖性分析时，或许需要使外部的管理者确定我们的系统符合他们的规则而且可能会需要完整的体系结构文档。

6.3 体系结构模式

作为一种表示、共享和复用软件系统知识的方法——模式的思想，现在得到了广泛的应用。有关面向对象设计模式这本书籍（Gamma等，1995）的出版引发了这种思想，这本书促进了其他类型模式的开发，例如机构设计模式（Coplien和Harrison，2004），可用性模式（Usability Group，1998），交互模式（Martin和Sommerville，2004），配置管理模式（Berczuk和Appleton，2002）等。体系结构模式在20世纪90年代以“体系结构风格”（Shaw和Garlan，1996）的名字

提出来，共有 5 卷关于面向模式的系统体系结构的系列手册在 1996 年到 2007 年间相继出版（Buschmann 等，1996；Buschmann 等，2007a；Buschmann 等，2007b；Kircher 和 Jain，2004；Schmidt 等，2000）。

在这一小节中作者首先介绍体系结构模式并简要地描述一种体系结构模式的选择，它是在不用类型系统中都适用的，如果读者希望了解更多关于模式及其使用的内容，可以参阅出版的模式手册。

我们可以把体系结构模式看做是对好的实践所做的格式化的抽象描述，它们已经在不同的系统和环境中多次尝试和测试过。所以，体系结构模式应当描述一种系统构成，这种构成在以往的系统是很成功的。体系结构模式还应该包括：什么时候这种模式适用，什么时候这种模式不适用，以及这种模式的优点和缺点等。

图 6-2 描述了众所周知的 MVC 模式，在许多基于 Web 的系统中，这种模型是交互管理的基础。这种格式化的模式描述包括模式的名字，一个简短的描述（伴有一个相关的图形模型），以及一个这种模式适用的系统类型的例子（可能也伴有一个图形模型）。我们还应该包括这种模式的应用时机和优缺点。在图 6-3 和图 6-4 中显示了与 MVC 模式相关的体系结构的图形模型，它们从不同的角度展现体系结构——图 6-3 是概念视图，而图 6-4 则给出了当该模式用于基于 Web 系统的交互管理时一个可能的运行时体系结构。

名字	MVC（模型 - 视图 - 控制器）
描述	将表示和交互从系统数据中分离出来。系统被设计成由 3 个彼此交互的逻辑组件组成：模型组件管理系统数据和在数据上的操作，视图组件定义和管理如何显示数据给用户，控制器组件管理用户的交互（例如，键按下，鼠标点击等），并传递这些交互给视图和模型。参见图 6-3
实例	图 6-4 说明了采用 MVC 模式的基于 Web 的应用系统的体系结构
使用时机	在数据有多个显示和交互方式的时候使用。也可在对未来数据的交互和表示需求不明朗的时候使用
优点	允许数据独立地改变，不影响表示，反之亦然。支持对相同数据的多种不同方式的表达，对某种表示的变更会传递到所有其他的表示
缺点	可能需要额外的代码，当数据模型和交互很简单时代码的复杂度相对较高

图 6-2 模型 - 视图 - 控制器（MVC）模式

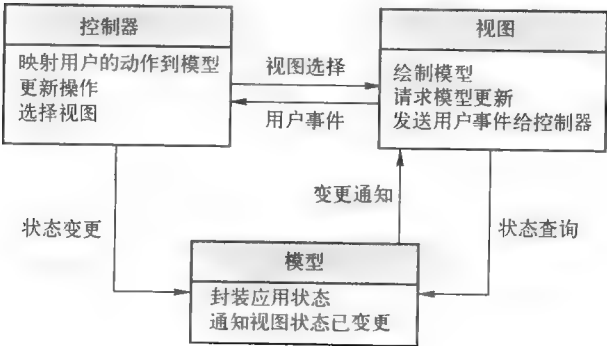


图 6-3 MVC 模式的组成

在一节中介绍清楚软件开发所有的通用模式是不可能的，因此，这里仅仅给出了一些精心挑选的常用模式的例子，它们能很好地遵循体系结构设计原则，更多通用的体系结构模式的例子放在了本书的网站上。

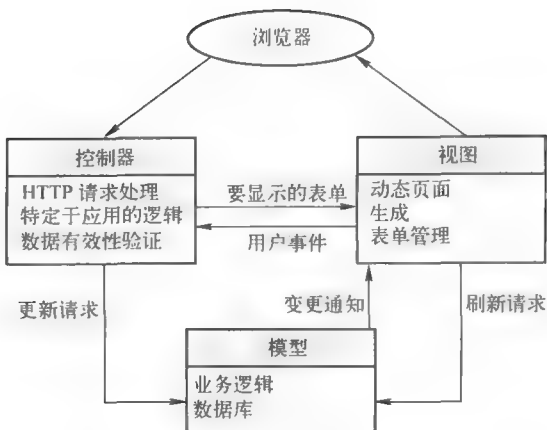


图 6-4 采用 MVC 模式的 Web 应用体系结构

6.3.1 分层体系结构

分离性和独立性的概念是体系结构设计的基础，因为分离性和独立性使得变更得到局部化。图 6-2 所示的 MVC 模式分离了系统的组成元素，允许它们独立地变更。例如增加一个新的视图或改变一个已有的视图，这些操作都可以在不改变模型底层数据的情况下完成。分层体系结构模式是实现分离性和独立性的另一个方式，图 6-5 显示了这种模式。这里，系统的功能被划分成几个独立的层次，每一层只依赖紧接的下一层所提供的服务和设施。

分层的方法支持系统的增量式开发。如一个层被开发完，该层提供的服务就可以被用户使用了。这个体系结构还是可改变的和可移植的。如果一层的接口被保留下来，这个层就能被另外一个对等层替换。当一层的接口改变或增加了新设施的时候，只有毗邻的层受影响。因为分层系统的抽象机依赖的是内层中的抽象机，因此，转换到其他机器上实现是比较容易的，此时只有内部与具体机器相关的层需要重新实现以适应不同的操作系统或数据库。

名称	分层体系结构
描述	将系统组织成分层结构，每一层中包含一组相关的功能。每一层提供服务给紧邻的上一层，因此最底层是有可能被整个系统所使用的核心服务。参见图 6-6
实例	存在于不同的图书馆中的共享版权文档的系统分层模型，参见图 6-7
使用时机	在已有系统的基础上构建新的设施时使用；当开发团队由多个分散的小团队组成，且每个小团队负责一层的功能时使用；或者是当系统存在多层信息安全性需求时使用
优点	允许在接口保持不变的条件下更换整个一层。在每一层中可以提供冗余服务（例如身份验证）以增加系统的可靠性
缺点	在具体实践中，在各层之间提供一个干净的分离通常是困难的，高层可能不得不直接与低层进行直接交互而不是间接通过紧邻的下一层进行交互。性能可能是个问题，因为服务请求会在每一层中被处理，所以需要多层解释

图 6-5 分层体系结构模式

图 6-6 是一个分为 4 层的体系结构的例子。最底层包括了系统支持软件，比较典型的是数据库和操作系统支持。再上一层是应用程序层，包括与应用功能相关的组件、可以被其他应用组件利用的实用工具组件等。第三层与用户界面管理相关，并提供用户的身份验证和授权。最上层提供用户界面设施。当然，分层的数量是随意的。图 6-6 中的任何一层都可以分为两层或更多层。

图 6-7 是分层体系结构模式应用于一个叫做 LIBSYS 的图书馆系统的示例，能够控制对一组大学图书馆中版权资料的电子访问。这个系统有 5 层，最底层是各个图书馆中独立的数据库。

我们能在图 6-17 中看到分层体系结构的另外一个例子。这个例子显示了前面章节中讨论的心理健康护理系统（MHC-PMS）的组织。

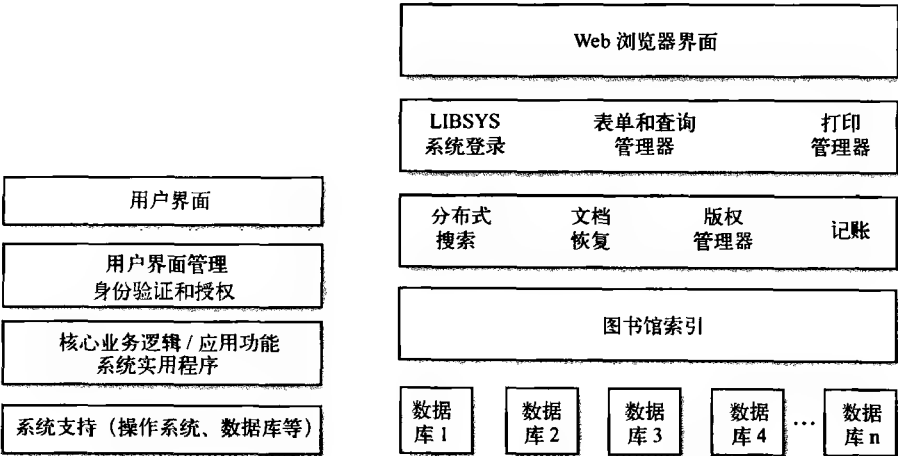


图 6-6 通用分层体系结构

图 6-7 LIBSYS 系统的体系结构

6.3.2 容器体系结构

分层体系结构和 MVC 模式都是模式的例子，是系统的概念组成。下一个例子是容器（Repository）模式（见图 6-8），描述的是一组交互组件如何共享数据。

名称	容器
描述	系统的所有数据在一个中央容器中管理，该中央容器可以被所有系统组件访问。组件间不是直接进行交互，它们只通过容器进行交互
实例	图 6-9 是 IDE 的一个实例，组件使用一个系统设计信息的容器。每个软件工具生成信息放入容器，然后被其他工具所使用
使用时机	当一个系统中所生成的大量信息需要持久保存时，可以使用该模式。也可以在数据驱动系统中使用该模式，每当在容器中收入数据时将触发一个动作或工具
优点	组件是独立的，它们无需知道其他组件的存在。一个组件的变更可以传播到所有的组件。所有的数据可以得到一致的管理（例如，在同一时间进行备份），因为它们是存放在同一个地方
缺点	容器是一个单个失败点，因而容器中的问题会影响整个系统。在组织所有通过容器进行的通信时会比较低效。将容器分布到多个计算机上会很困难

图 6-8 容器模式

大多数使用大量数据的系统都是围绕共享数据库或容器来组织的。因此，这个模型适合于数据是由一个组件产生而由其他组件使用的情形。这种类型的系统例子包括指挥和控制系统、管理信息系统、CAD 系统和软件的交互开发环境等。

图 6-9 说明了一个可能会用到容器的情形。该图显示了一个包含不同工具来支持模型驱动开发的 IDE 系统。在这个例子中，容器或许就是一个能跟踪软件变更并允许回滚到先前版本的版本控制环境（如将在第 25 章中所讨论的）。

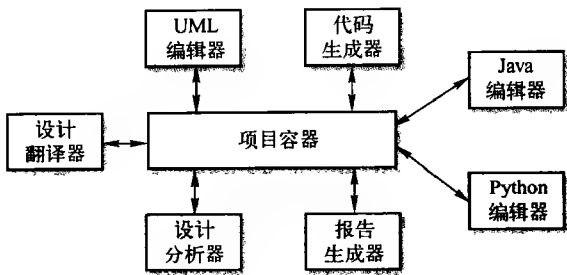


图 6-9 IDE 系统的容器体系结构

把所有适合使用容器的工具组织起来是共享大量数据的一种高效方式。这就不需要显式地把数据从一个组件传送到另一个组件。然而，组件一定要围绕一个约定好了的容器数据模型运行。这不可避免地要在每个工具的特定需求之间做出妥协。若一个新组件的数据模型与该模型有冲突，那么要想将它集成到该系统中来可能很困难或是不可能的。实际上，将容器分布到多台机器上可能是困难的。在逻辑上，虽然将集中式容器分布到不同的机器上是可能的，但这样做会引起数据冗余和不一致性的问题。

如图 6-9 中所示，容器是被动的，对它的控制是组件的职责。另外一种方法源于人工智能，即使用所谓的“黑板”模型，当有特别的数据可用时，就会主动通报组件。当容器数据的结构组织得不是很好时该方法比较合适。到底激活哪个工具要视对数据的分析结果而定。该模型由 Nii (1986) 给出，Bosch (2000) 分析了该风格与系统质量属性之间的关系。

6.3.3 客户机 - 服务器体系结构

容器模式与系统的静态结构有关，但是不能展现出它的运行组织。下一个例子说明了一种常用的分布式系统的运行时组织。图 6-10 描述了客户机 - 服务器模式。

名称	客户机 - 服务器
描述	在客户机 - 服务器体系结构中，系统的功能是以服务的形态存在的，每一个服务来自于某个单独的服务器。客户机是那些使用服务和访问服务器的用户
实例	图 6-11 是一个电影和视频/DVD 资料库，示例是以客户机 - 服务器形式存在的系统
使用时机	当需要从很多地点访问共享数据时使用。因为服务器可以复制，所以也可以在系统负载经常变化时使用
优点	该模型的主要优点是服务器可以分布到网络上。一般性的功能（例如打印服务）可以被所有的客户机使用，但并不需要被所有的服务所实现
缺点	每个服务是单个失败点，所以对阻止拒绝服务攻击或服务器失败缺乏免疫性。性能也可能是无法预知的，因为它依赖于网络也依赖于系统。当服务器属于不同的机构时，也存在管理方面的问题

图 6-10 客户机 - 服务器模式

一个采用客户机 - 服务器模式的系统是由一个服务集合和相关的服务器以及访问和使用这些服务的客户机组织起来的。这个模型的主要组成部分是：

- 1. 一组给其他组件提供服务的服务器。服务器的例子包括：打印服务器，提供打印服务；文件服务器，提供文档管理服务；编译服务器，负责对程序的编译服务。
- 2. 一组向服务器请求服务的客户机。一个客户机程序通常有多个实例，可以在不同的计算机上并发执行。
- 3. 一个连接客户机和服务器的网络。绝大多数客户机 - 服务器系统实现为分布式系统，通

过互联网的协议连接在一起。

客户机-服务器体系结构经常被认为是分布式系统体系结构，但是运行在分散服务器上的独立服务的逻辑模型可以在单个计算机上实现。此外，更重要的好处是分离性和独立性。服务和服务器可以改变而不会影响系统其他部分。

客户机必须知道可用的服务器的名字及它们所提供的服务。反之，服务器没有必要知道客户机的身份以及到底有多少客户机在访问它们的服务。客户机通过远程过程调用来获取服务器提供的服务，远程过程调用使用一个请求-回答协议，比如在 WWW 上使用的 http 协议。本质上，客户机向服务器提出请求，然后等待直到它收到回答为止。

建立在客户机-服务器模型上的一个系统例子如图 6-11 所示。这是一个提供电影和图片库的多用户的基于 Web 的系统。在这个系统中，有管理和放映不同类型媒体的多个服务器。视频信号需要快速、同步地传输，但分辨率相对较低。它们是以压缩的形式存储的，所以视频服务器需要对于各种不同的格式处理视频压缩和解压缩。静态图片必须保持高分辨率，所以将它们单独放在一个服务器上是比较妥当的。

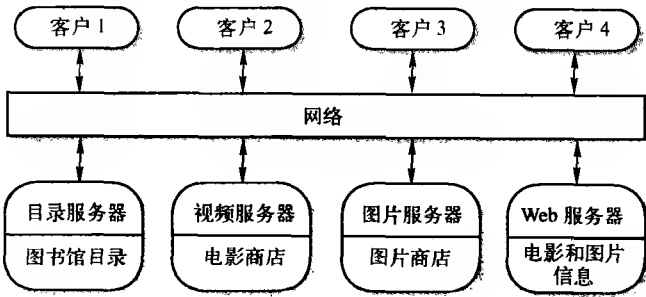


图 6-11 电影资料库的客户机-服务器体系结构

要求目录能够支持各种查询，能与包含电影和视频片段的 Web 信息系统链接，并能与支持发售图片、电影和视频片段的电子商务系统保持链接。客户机程序只是对访问这些服务提供一个集成的用户界面（用 Web 浏览器构造）。

客户机-服务器模型的最大优势在于它是一个分布式体系结构。由许多分布式处理器构成的网络系统可提供更有效的使用。在这样的系统中，添加一台服务器并将它与系统其余部分集成在一起，或透明地更新服务器而不影响系统的其他部分是很容易的。第 18 章将更详细地讨论分布式体系结构，包括客户机-服务器体系结构和分布式对象体系结构。

6.3.4 管道和过滤器体系结构

最后一个体系结构模式是管道和过滤器模式，见图 6-12。这是一个系统运行时组织的模型，在这个模型中，函数转换处理输入并产生输出。数据从一个处理单元流到另一个处理单元，每经过一个单元就做一次变换。输入数据流经过这些变换直到转换为输出。这些转换可能顺序地或并行地执行，数据加工可以是一项一项地处理，也可以成批处理。

“管道和过滤器”的名字最早出自 Unix 系统，在 Unix 系统中在链接进程时可能会用到“管道”。这些管道能从一个进程到另一个进程传递文本流。遵照这个模型的系统可以通过组合 Unix 命令、使用管道和 Unix Shell 控制工具来实现。“过滤器”这个词的使用很形象地描述了数据从输入到输出这样一个过程。

名称	管道和过滤器
描述	系统中数据的处理是这样组织的，每个处理组件（过滤器）都是分离的并执行某个类型的数据转换。数据流（如在一个管道中）从一个组件流向另一个组件
实例	图 6-13 是用于处理票据的管道和过滤器系统的例子
使用时机	一般应用在数据处理应用中（批处理和事务处理），一些不同的阶段处理输入数据，并产生相应的输出
优点	易于理解并支持变换的复用。工作流风格与很多业务处理体系结构很匹配。通过添加变换的方式进行进化是很显然的。可以实现为顺序的系统，也可以实现为并发的系统
缺点	在通信变换间所传输的数据格式必须协商好。每个变换必须解析它的输入并写成约定的格式输出。这增加了系统的负荷，意味着不可能复用使用不兼容数据结构的函数变换

图 6-12 管道和过滤器模式

自计算机首次用于自动数据处理以来，数据流模型的多种形式就被广泛使用。当对数据的转换是顺序进行时，这种管道和过滤器体系结构就变成了批处理模型，对于像票据处理系统这样的一类数据处理系统，这是一个非常普通的模型。嵌入式系统的体系结构也可能以进程流水线的形式组织，每一个进程都是当前正在执行的。这种模型在嵌入式系统中的应用将会在第 20 章中讨论。



控制的体系结构模式

关于系统中控制的常用组织方法，由专门的体系结构模式来表达。这些包括：集中式控制，基于一个组件调用另一个组件，以及基于事件的控制（系统响应来自外部的的事件）。

<http://www.SoftwareEngineering-9.com/Web/Architecture/ArchPatterns/>

这种类型的系统体系结构用于批处理应用的一个例子如图 6-13 所示。一个机构给其客户开出账单。每星期将付款与账单核对一次，若账单已经支付，则开出收据；若在规定的时间内尚未支付，则发出支付提醒。

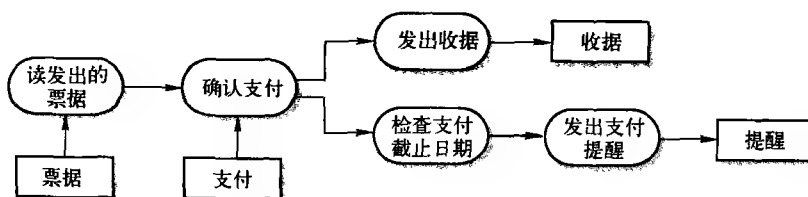


图 6-13 管道和过滤器体系结构的实例

交互式系统很难用管道和过滤器模型来描述，因为有需要处理的数据流的要求。尽管对于简单本文的输入和输出，用这个方法建模是可以的，但对图形化的用户界面来说，有比较复杂的输入/输出格式和基于事件（如鼠标点击或菜单选择）的控制策略，将这些翻译成与这个数据流模型兼容的格式是相当困难的。

6.4 应用体系结构

应用系统的目的是要满足某些商业和机构的需要。所有的业务都是共通的——需要雇佣人员、开出发票、记账等。同一个部门中的业务通常使用部门特定的应用系统。因此，除了一般的业务功能，所有的电话公司需要系统去连通线路、管理它们的网络、向客户开单据等。因此，

这些业务所使用的应用系统也有很多共同点。

这些共性导致了用来描述特殊类型软件系统的结构和组成的软件体系结构的发展。应用体系结构封装了一类系统的基本特征。例如，在实时系统中，就有不同系统类型的通用的体系结构模型，像数据采集系统或监控系统。尽管这些系统实例在细节上有所不同，但是当开发同一类型的新系统时，我们是可以用通用体系结构架构的。

当开发新的系统时应用体系结构往往要重新实现，但是对大多数商务系统，应用的复用有可能不需要重新实现。我们可以看到出自 SAP 和 Oracle 公司的企业资源规划（ERP）系统以及为特殊应用所提供的垂直软件包（COTS）的增长。在这些系统中，我们可以发现存在一个通用的配置，通过调整它们来产生专门的业务应用。作为一个例子，供应链管理系统可以调整以面向不同类型的供应商、商品以及合同管理。



应用体系结构

有多个应用体系结构的实例在本书的网页上。它们包括有关批处理系统、资源分配系统、基于事件的编辑系统的描述。

<http://www.SoftwareEngineering-9.com/Web/Architecture/AppArch/>

作为一名软件设计人员，我们可以以多种方式来使用这些应用体系结构模型：

1. 作为体系结构设计过程的一个起点 如果我们不熟悉正在开发的应用类型，我们就可以将我们的初始设计建立在一般体系结构基础上。当然，这些都必须针对被开发的专门系统进行特殊处理，但不管怎样，它们都是我们设计的一个很好的开始。

2. 作为设计检查列表 如果我们已经为一个应用系统完成了系统体系结构设计，我们可以将它与一般应用体系结构进行比较，来检查我们的设计是否和一般体系结构是一致的。

3. 作为对开发团队工作的组织方式 应用体系结构寻找系统体系结构的稳定的结构特征，而且，在很多情况下，是可能并行地开发这些的。我们可以分派任务给团队成员，在同一个体系结构框架下实现不同的组件。

4. 作为评估组件以便复用的手段 如果我们拥有组件，我们就可以复用它。在希望复用时，可以将它们与一般结构做比较来看在应用体系结构中是否有类似的组件。

5. 作为交流应用类型的词汇 如果我们在讨论一个特殊的应用或者试图比较几个具有相同类型的应用时，那么，我们就可以使用在一般体系结构中所有的概念来谈论应用了。

有很多种类的应用系统有时看起来是很不一样的。但是，我们发现很多表面上看风格迥异的两种应用本质上却是基本相同的。通过以下两种应用类型的体系结构来说明这一点：

1. 事务处理应用 事务处理应用是以数据库为中心的，处理来自用户对信息的请求并更新数据库的数据。这些是最为平常的一类交互式业务系统类型。对它们的组织要能够保证用户之间是不会相互干扰的，数据库的整体性是得到保障的。这种类型系统包括交互式银行系统、电子商务系统、信息系统和预订系统。

2. 语言处理系统 语言处理系统是这样一类系统，用户意图用形式化语言（比如 Java）来表达的系统。语言处理系统将这种语言处理成一种内部格式，然后解释这种内部表示。我们最为熟悉的语言处理系统就是编译器了。它将高层语言翻译成机器代码。然而，语言处理系统也使用在很多其他系统中，比如在数据库中作为对命令语言的翻译，在信息系统中和在标记语言比如 XML（Harold 和 Means, 2002; Hunter 等, 2007）中。

之所以选择这些特殊类型系统是因为许多基于 Web 的商务系统是事务处理系统，而且所有

的软件开发都依赖语言处理系统。

6.4.1 事务处理系统

事务处理系统 (TP) 是设计用来处理用户对数据库信息查询或者请求对数据库更新的 (Lewis 等, 2003)。从技术角度讲, 数据库事务是一个操作序列, 每个这样的操作可以看成是一个单元 (原子单元)。事务中的所有操作都必须在数据库永久改变之前完成。这确保在事务中操作失败不会导致数据库的不一致性。

从用户的角度看, 事务是任何一个相关操作的序列, 这些操作能达到某个目的, 比如“查询从伦敦飞往巴黎的班机的时间”。如果用户事务不要求数据库发生改变, 那么就没有必要把它作为数据库事务。

事务的例子可以是客户请求使用 ATM 机从银行账户上提取现金。这包括: 获得客户账户的详细信息, 检查账户余额, 修改支取现金后的账户余额, 发送指令给 ATM 机输出现金。除非所有上述步骤都已经完成, 否则, 事务是没有完成的, 客户账户数据库也是不会发生改变的。

事务处理系统总是交互式系统, 用户异步地提出对服务的请求。图 6-14 给出了事务处理应用的概念体系结构。首先, 用户通过 I/O 处理组件向系统发出请求。请求会被应用相关的逻辑进行处理。事务得以创建并传递给事务管理器, 事务管理器嵌入在数据库管理系统内。在事务管理器对事务的正确完成检查完毕后, 传递一个信号给应用, 报告处理已经完成。



图 6-14 事务处理应用的体系结构

事务处理系统可以组织成“管道和过滤器”的结构, 分别由系统组件负责输入、处理和输出。例如, 允许客户在 ATM 机上查询账户余额和提取现金的银行系统。这个系统由两个相互协作的软件子系统构成, 即 ATM 软件和位于银行数据库服务器上的账户处理软件。输入输出组件实现为 ATM 机上的软件, 而处理组件是位于银行数据库服务器上的。图 6-15 显示了这个系统的体系结构, 说明了输入、处理和输出组件的功能。

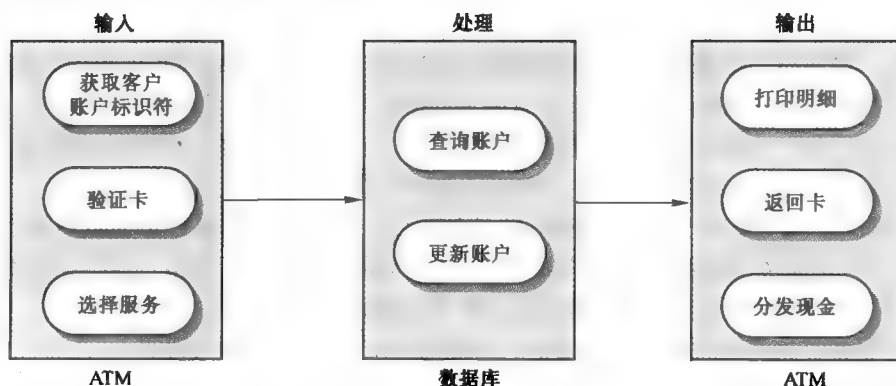


图 6-15 ATM 机系统的软件体系结构

6.4.2 信息系统

所有涉及与共享数据库交互的系统都可以看成是基于事务的信息系统。信息系统允许对一个大信息库进行适当的访问。这些大数据库例子有图书馆书目库、航班时刻表、医院的病人记录

等。越来越多的信息系统是通过浏览器访问的基于 Web 的系统。

图 6-16 是一个信息系统的一般模型。系统是用分层方法（在 6.3 节中讨论过），顶层支持用户界面，底层是系统数据库。用户通信层处理所有来自用户界面的输入和输出，信息检索层包括应用相关的访问和更新数据库的逻辑。如我们将在后面看到的，这个模型中的层能直接映射到基于 Internet 系统的服务器上。

作为分层模型的一个实例，图 6-17 给出了 MHC-PMS 系统的体系结构。我们知道这个系统维护并管理那些具有心理问题前来咨询专家的病人的详细信息。在模型的每一层中都增加了一些详细内容，找出了用于支持用户通信、信息检索和访问的组件：

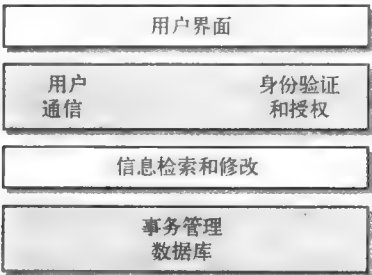


图 6-16 分层的信息系统体系结构

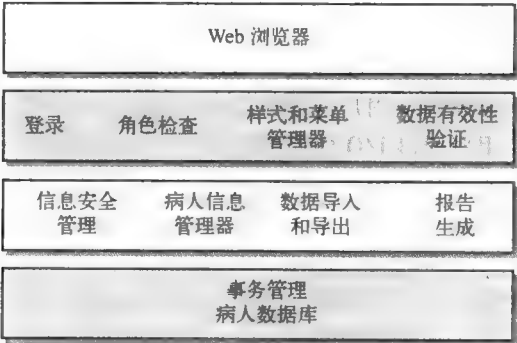


图 6-17 MHC-PMS 系统的体系结构

1. 最上层负责实现用户接口。在这个例子中，UI 是用 Web 浏览器实现的。
2. 第二层提供用户接口的功能，这是通过 Web 浏览器来传送的。该层包括允许用户登录系统的组件和确保他们使用的操作符合他们身份的检验组件。这一层还包括显示信息给用户的表格和菜单管理组件以及核对信息一致性的数据确认组件。
3. 第三层实现系统的功能，并提供：实现有关系统信息安全，病人信息的建立和更新，从其他数据库载入和导出病人数据，以及报告生成器建立管理报告。
4. 最后，最底层是使用商用的数据库管理系统建立的，它提供事务管理和持久的数据存储。

信息和资源管理系统现在通常是基于 Web 的系统，用户接口是使用 Web 浏览器实现的。例如，电子商务系统是一种基于互联网的资源管理系统，它可以接受电子的商品或服务的订单，然后安排将商品或服务递送给客户。在电子商务系统中，应用相关层包括额外的功能，支持所谓的“购物车”，即用户可以在不同的事务中购买多个项目，然后在一个事务中完成对所有项目的支付。在这些系统中服务器的组成可以映射为图 6-16 所示的四层通用模型。这些系统总是实现为多层的客户机/服务器体系结构，如我们在第 18 章中所看到的那样：

1. Web 服务器负责对所有用户的通信，用户接口是 Web 浏览器实现的。
 2. 应用服务器负责实现应用相关的逻辑，也包括信息存储和检索请求。
 3. 数据库服务器将信息从数据库中移入和移出，并且处理事务管理。
- 使用多服务器允许高吞吐量并且使得在每分钟内处理数百个事务成为可能。随着需求的增加，可以通过在每一层增加服务器来处理额外的相关进程。

6.4.3 语言处理系统

语言处理系统把自然语言或人工语言翻译成该类语言的其他表示，对于编程语言可能会执行产生的代码。在软件工程中，编译器把人工的程序语言翻译成机器码。其他语言处理系统把 XML 数据描述翻译成命令来查询数据库，或者翻译成替代 XML 的表示。自然语言处理系统能把一种自然语言翻译成另一种自然语言。比如法语翻译成挪威语。

图 6-18 给出了编程语言的一种语言处理系统的可能的体系结构。源语言指令定义了将要执行的程序，翻译器会转换这些称为抽象机指令。这些指令然后由另一个组件解释，它首先取指令，然后再（在必要的时候）使用来自环境中的数据去执行它们。该过程的输出是在输入数据上的指令的解释结果。

当然，对于很多编译器，解释器是一个处理机器指令的硬件单元，抽象机是一个真处理器。然而，对于动态类型的语言比如 Python，解释器是一个软件组件。

程序语言编译器是更一般化的编程环境的一部分，它具有一般的体系结构，包括以下组件：

1. 词法分析器，将一个输入的语言记号转换为内部形式；
2. 符号表，保持实体名字相关的信息（变量、类名字、对象名字等），实体是正在翻译的文本中所使用的实体；
3. 语法分析器，它检查正在翻译的语言的语法。它使用相应语言所定义的语法并建立语法树；
4. 语法树，它代表待编译程序的内部结构；
5. 语义分析层，它使用来自语法树和符号表的信息来检查输入的语言文本的语义正确性；
6. 代码生成器，它在语法树中穿行并生成抽象机代码。

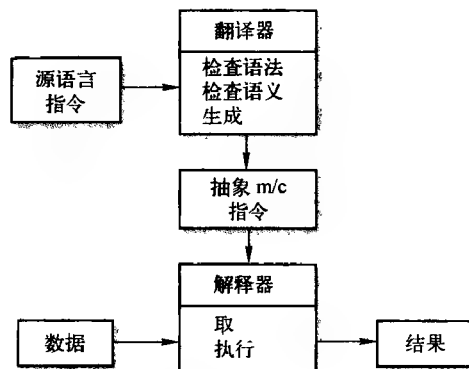


图 6-18 语言处理系统的体系结构



参考体系结构

参考体系结构捕捉领域内的系统体系结构的重要特征。虽然本质上它们包括应用体系结构中的每一样东西，但是事实上，单独一个应用不太可能包括参考体系结构中的所有特征。参考体系结构的主要目的在于评估和比较设计提议，以及在领域内部培训体系结构的知识。

<http://www.SoftwareEngineering-9.com/Web/Architecture/RefArch.html/>

也还会有其他一些组件，这些组件可能用于从所生成的机器代码中分析和变换语法树来提高效率和降低冗余。在其他类型语言处理系统中，比如自然语言翻译器将有附加的组件比如字典组件，所生成的代码实际上是用其他语言所表示的输入文本。

在语言处理系统中可能会用到的还有另外可用的体系结构模式（Garlan 和 Shaw，1993）。编译器可以用容器和管道过滤器的复合模型实现。在编译器的体系结构中，符号表是共享数据容器。在词法分析、语法和语义分析阶段组成一个串行结构，如图 6-19 所示，它们之间的通信是通过共享符号表实现的。

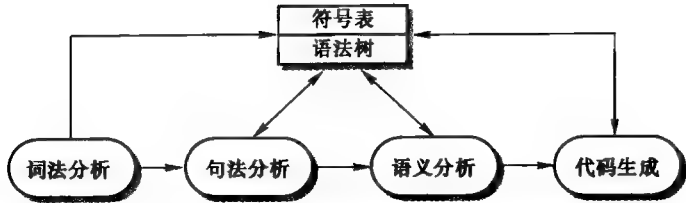


图 6-19 编译器的管道和过滤器体系结构

语言编译的管道和过滤器模型对于程序的编译和执行无需用户交互的批处理环境是很有效的。例如，把一个 XML 文件翻译成另一个文件。而对于集成了其他语言处理工具，比如结构化编辑系统、交互式调试工具或者是程序的格式转换器，它就不那么有效了。在这种情况下，来自一个组件的变化需要立即反映到其他的组件中来。因此，系统最好组织成容器的模型，如图 6-20 所示。

图 6-20 说明了一个语言处理系统是如何成为集成化编程支持工具的一部分的。在这个例子中，符号表和语法树是作为中央信息容器的。工具或小程序通过它来实现通信。其他的原先可能嵌入在工具内部的信息，比如语法定义和程序的输出格式定义等，已经被提取出来并放入这个容器中。因此，语法制导的编辑器能够在程序输入时检查程序的语法是正确的，而程序格式转换器能够以一种十分方便阅读的格式创建程序清单。

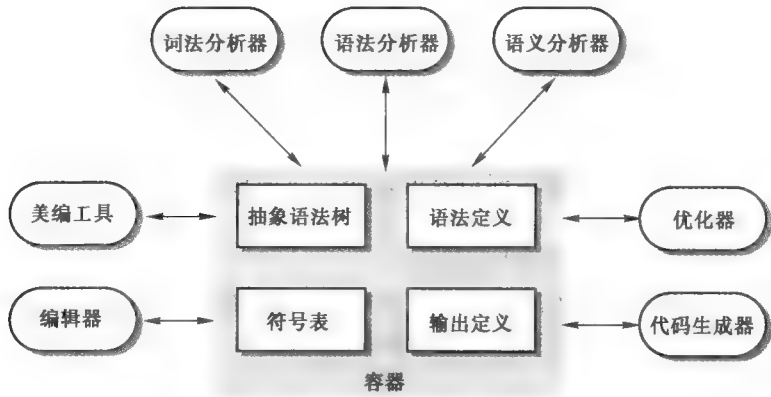


图 6-20 语言处理系统的容器体系结构

要点

- 软件体系结构是有关软件系统如何组织的描述。系统的性质，比如性能、信息安全性和可用性，都受到所使用体系结构的影响。
- 体系结构设计决策包括对应用类型的决策、系统分布的决策、所使用的体系结构风格的决策，以及对体系结构应该如何文档化和评估的决策。
- 体系结构可能会从许多不同的视角和视图被文档化。可能的视图包括概念视图、逻辑视图、进程视图、开发视图和物理视图。
- 体系结构模式是复用通用的（一般的）系统体系结构知识的一种方法。它描述体系结构，解释这样的体系结构的使用时机以及它的优缺点。
- 应用较多的体系结构模式有 MVC、分层体系结构、容器结构、客户机 - 服务器结构，以及管道和过滤器结构。
- 应用系统体系结构的一般模型能帮助我们理解应用的运作，比较相同类型的应用，验证系

应用系统设计的有效性并能达到对大粒度组件的复用。

- 事务处理系统是交互式系统，允许数据库中的信息被很多远程用户访问和修改。信息系统和资源管理系统是事务处理系统的例子。
- 语言处理系统用来将文本从一种语言翻译成另一种语言，来执行输入语言所定义的指令。它们包括一个翻译器和一个执行生成的语言的抽象机。

进一步阅读材料

《Software Architecture: Perspectives on an Emerging Discipline》 这是第一本关于软件体系结构的书，书中有关于不同体系结构风格的很好的讨论（M. Shaw and D. Garlan, Prentice-Hall, 1996）。

《Software Architecture in Practice, 2nd ed》 该书是对软件体系结构的实际讨论，它不是对体系结构设计的好处的宣扬，而是给出了一个清晰的工作原理，说明为什么体系结构是重要的（L. Bass, P. Clements 和 R. Kazman, Addison-Wesley, 2003）。

“The Golden Age of Software Architecture” 这篇论文纵览了软件体系结构从 20 世纪 80 年代开始到目前使用的发展过程。其中技术内容不多，却是一篇有趣的历史性概述。（M. Shaw 和 P. Clements, IEEE Software, 21 (2), March-April 2006.）<http://dx.doi.org/10.1109/MS.2006.58>。

《Handbook of Software Architecture》 这是一部 Grady Booch 正在编写的著作，Grady Booch 是软件体系结构早期的传道士。他已经记载了一系列的软件系统的体系结构，所以我们看到的绝对不是单独的学术上的抽象概念。网上提供并且将要出版。<http://www.handbookofsoftwarearchitecture.com/>。

练习

- 6.1 当描述一个系统时，解释为什么我们必须在应用需求完成之前设计系统体系结构。
- 6.2 假如一个不懂技术的管理者要求我们准备并且提交一份报告来证明为一个新项目雇佣一个系统架构师是有道理的。在我们的报告中用简要文字列出要点。当然，我们必须解释什么是系统体系结构。
- 6.3 解释为什么在设计可用性和信息安全需求都是最为重要的非功能性需求的系统的体系结构时会发生设计冲突。
- 6.4 画图说明以下系统体系结构的概念视图和过程视图：
 - 地铁乘客使用的自动售票系统。
 - 电脑控制的视频会议系统，系统包括视频、音频以及同时显示给多个参与者的电脑数据。
 - 机器人清理地板系统，这个系统想要清理相对干净的场所，比如走廊。清理者必须能够觉察到墙面和其他障碍物。
- 6.5 解释为什么在为一个大系统体系结构的时候我们通常会使用多种体系结构模式。除去本章中讨论的关于模式的知识，在设计大型系统的时候还有其他有用的知识吗？
- 6.6 针对在线出售和发布音乐的系统（比如 iTunes）推荐一种体系结构，这种体系结构的基础是什么结构模型？
- 6.7 解释我们将如何使用 CASE 环境的参考模型（在本书的网页上提供的）来比较不同程序语言（比如 Java）提供商提供的 IDE。
- 6.8 使用这里介绍的语言处理系统的一般模型，设计接收自然语言命令并能将它们翻译成如 SQL 这样的语言的数据库查询语句的系统的体系结构。
- 6.9 使用如图 6-16 所示的信息系统基本模型，说明允许用户观察航班到达和离开某个空港信息的信息系统的组件组成。
- 6.10 是否应该存在一位单独的软件架构师的职业，它的角色是独立地与客户一起设计软件系统的体系结构？一个单独的软件公司将要实现系统，设立这样的一个职业的困难会是什么？

参考书目

- Bass, L., Clements, P. and Kazman, R. (2003). *Software Architecture in Practice*, 2nd ed. Boston: Addison-Wesley.
- Berczuk, S. P. and Appleton, B. (2002). *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Boston: Addison-Wesley.
- Booch, G. (2009). 'Handbook of software architecture'. Web publication.
<http://www.handbookofsoftwarearchitecture.com/>.
- Bosch, J. (2000). *Design and Use of Software Architectures*. Harlow, UK: Addison-Wesley.
- Buschmann, F., Henney, K. and Schmidt, D. C. (2007a). *Pattern-oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. New York: John Wiley & Sons.
- Buschmann, F., Henney, K. and Schmidt, D. C. (2007b). *Pattern-oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. New York: John Wiley & Sons.
- Buschmann, F., Meunier, R., Rohnert, H. and Sommerlad, P. (1996). *Pattern-oriented Software Architecture Volume 1: A System of Patterns*. New York: John Wiley & Sons.
- Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. (2002). *Documenting Software Architectures: Views and Beyond*. Boston: Addison-Wesley.
- Coplien, J. H. and Harrison, N. B. (2004). *Organizational Patterns of Agile Software Development*. Englewood Cliffs, NJ: Prentice Hall.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley.
- Garlan, D. and Shaw, M. (1993). 'An introduction to software architecture'. *Advances in Software Engineering and Knowledge Engineering*, 1 1–39.
- Harold, E. R. and Means, W. S. (2002). *XML in a Nutshell*. Sebastopol, Calif.: O'Reilly.
- Hofmeister, C., Nord, R. and Soni, D. (2000). *Applied Software Architecture*. Boston: Addison-Wesley.
- Hunter, D., Rafter, J., Fawcett, J. and Van Der Vlist, E. (2007). *Beginning XML*, 4th ed. Indianapolis, Ind.: Wrox Press.
- Kircher, M. and Jain, P. (2004). *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. New York: John Wiley & Sons.
- Krutchén, P. (1995). 'The 4+1 view model of software architecture'. *IEEE Software*, 12 (6), 42–50.
- Lange, C. F. J., Chaudron, M. R. V. and Muskens, J. (2006). 'UML software description and architecture description'. *IEEE Software*, 23 (2), 40–6.
- Lewis, P. M., Bernstein, A. J. and Kifer, M. (2003). *Databases and Transaction Processing: An Application-oriented Approach*. Boston: Addison-Wesley.
- Martin, D. and Sommerville, I. (2004). 'Patterns of interaction: Linking ethnomethodology and design'. *ACM Trans. on Computer-Human Interaction*, 11 (1), 59–89.
- Nii, H. P. (1986). 'Blackboard systems, parts 1 and 2'. *AI Magazine*, 7 (3 and 4), 38–53 and 62–9.

Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F. (2000). Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects. New York: John Wiley & Sons.

Shaw, M. and Garlan, D. (1996). Software Architecture: Perspectives on an Emerging Discipline. Englewood Cliffs, NJ: Prentice Hall.

Usability group. (1998). 'Usability patterns'. Web publication.
<http://www.it.bton.ac.uk/cil/usability/patterns/>.

设计与实现

目标

本章的目标是介绍运用 UML 进行面向对象软件设计，并强调重要的实现问题。读完本章，你将了解以下内容：

- 理解一般的面向对象设计过程中最重要的活动；
- 理解用于记录面向对象设计的一些不同的模型；
- 知道设计模式的思想，以及如何将其用于设计知识和经验的复用；
- 了解实现软件时需要考虑的关键因素，包括软件复用和开源开发。

软件设计和实现是软件工程过程中的一个阶段，在此阶段开发出可执行的软件系统。对简单的系统来说，软件设计和实现就是软件工程的全部，其他的所有活动都融汇在这个过程中。但是，对于大型的系统，软件设计和实现只是一系列软件工程过程（如需求工程、检验和有效性验证等）中的一个。

软件设计和实现活动总是交叉进行的。软件设计是创造性活动，我们可以基于客户的需求识别系统组件及其关系。实现是将设计转变为程序的过程。有时有一个独立的设计阶段完成设计建模和文档化。有时设计在程序员的脑中完成或者粗略地在白板或草稿纸上绘制草图来完成。设计是关于如何解决问题的活动，因此通常有一个设计过程。但是，用 UML 及其他描述语言描述设计并不总是必要或者合适的。

设计和实现是紧密相连的，因此我们通常需要在设计的过程中考虑到实现的因素。例如，对于用面向对象语言（如 Java 或 C#）开发来说，用 UML 来记录设计是很好的方法。但是对于用动态语言（如 Python）开发，UML 则不是那么有用；如果系统的实现采用的方法是配置现成的包，那么 UML 就一点作用也没有了。第 3 章中已经讨论过，敏捷方法通常用非正式的草图设计形式，因此赋予编程人员很多决定权。

软件项目早期要决定的最重要的实现决策之一就是应该购买还是构建应用软件。目前在很广泛的应用领域中都可以购买商业现货系统（COTS），然后根据用户需求进行调整和裁剪。举例来说，我们想开发一个医疗记录系统，那么可以购买医院中已经使用的程序包。采用这种方法的开发成本会更低，相对于常规的用编程语言开发的方法，开发速度也更快。

采用这种方式开发应用程序时，设计过程要集中考虑如何配置这些功能，使得系统可以满足现有需求。通常不需要为系统做设计模型，如系统对象及其交互的模型。第 16 章将对基于 COTS 的方法进行论述。

假设本书的读者大多具备编程和实现的经验，这些经验是我们学习编程并精通某种语言（如 Java 或 Python）所需要的。你可能已经学到了编程语言中良好的编程习惯，以及如何调试开发的程序。因此，在此并不涉及编程的话题。本章有如下两个目标：

1. 在面向对象软件设计的实践中，如何实现系统建模和体系结构设计（已在第 5 章和第 6 章中讨论）。
2. 介绍有关编程的书中并未涉及的重要实现问题。包括软件复用、配置管理和开源开发。

由于开发的平台很多，本章不着重面向任何特定的编程语言或实现技术。因此，采用 UML

来描述所有的例子，而不是用某种特定的编程语言，如 Java 或 Python。



结构化设计方法

结构化设计方法建议，软件设计应该以一种有条理的方法加以解决。设计一个系统需要遵从该方法的步骤并逐步细化系统的设计到各个细节层次。在 20 世纪 90 年代，出现了多个面向对象设计的好方法。然而，最常使用的方法的开创者们聚集在一起并提出了 UML，对用于不同方法的符号进行了统一。

现在，讨论的焦点不是方法，而是过程，在此，设计被看成总的软件开发过程的一部分。Rational 统一过程（RUP）是通用开发过程的一个典范。

<http://www.SoftwareEngineering-9.com/Web/Structured-methods/>

7.1 利用 UML 进行面向对象设计

面向对象系统是由一组相互交互的对象所组成，这些对象维护它们自己的局部状态并对这些状态提供操作。对状态的表示是私有的，不能被外部对象直接访问。对象设计过程包括设计对象类和这些类之间的关系。这些类定义了系统中的对象和它们的交互。当设计被实现为一个执行系统的时候，对象从它们的类定义中被动态创建出来。

面向对象的系统比起用功能方法开发出来的系统更容易改变。对象中包含数据及处理这些数据的操作，因此它们可以被当做一个独立的实体去理解和修改，变更对象的实现或者往对象中添加服务不应该影响系统中其他的系统对象。因为对象与真实事物相关，因此，在真实世界中的实体（例如，硬件组件）和系统中的控制对象之间存在清晰的映射。这一点增进了可理解性，进而也增进了设计的可维护性。

开发系统设计，由概念设计转变为详细的面向对象的设计，需要完成如下几点：

1. 了解并定义上下文和与系统的外部交互。
2. 设计系统体系结构。
3. 识别出系统中的主要对象。
4. 开发设计模型。
5. 定义对象接口。

正如所有的创意性活动一样，设计并不是一个清晰、顺序的过程。设计是一个获得设计理念，提出方案，并随着信息的收集来精练方案的过程。当问题出现的时候，我们将不可避免地回溯和重试这个过程。有时需要详细地探究自己的选择是否可行，而有时又会忽略细节直到过程的后期阶段。因此，不能用一张简单的图来描述这个过程，因为图的形式容易让人误认为设计是一个顺序的过程。而事实上，上述所有活动都是交织在一起并互相影响的。

下面通过第 1 章中介绍的野外气象站系统这个案例的设计部分来讲解这些活动。野外气象站部署在远程的区域，每个气象站记录当地的气象信息，并通过卫星链接将这些信息定期传输给气象信息系统。

7.1.1 系统上下文与交互

任何软件设计过程的第一阶段都是去了解待开发软件和外部环境之间存在的关系。必须做出决策，以确定如何提供系统所需要的功能以及如何构成系统以便它能有效地与环境进行通信。理解上下文还对系统边界的建立有所帮助。

设置系统上下文边界有助于我们决定什么特征是在建系统中实现的, 哪些特征是存在于其他关联系统中的。在此情况下, 你需要确定功能在负责所有气象站的控制系统和气象站自身的嵌入式软件两者间的分布。

系统上下文模型和交互模型可以互为补充, 呈现出系统和环境之间的关系:

1. 系统上下文模型是一个结构模型, 描述开发中的系统所处环境中的其他系统。
2. 交互模型是一个动态模型, 表明系统在运行中是如何与环境交互的。



气象站系统用例

报告天气——发送气象数据至气象信息系统

报告状态——发送状态信息至气象信息系统

重新启动——如果气象站系统已关闭, 则重新启动系统

关机——关闭气象站

重配置——重新配置气象站软件

节电——将气象站设定为节电模式

远程控制——向任意气象站子系统发送控制指令

<http://www.SoftwareEngineering-9.com/Web/WS/Usecases.html>

系统的上下文模型可以用关联来表示。关联可以简单地表示出关联所涉及的实体之间存在的关系, 此时确定了关系的性质。因此我们可以用简单的框图表示系统环境, 以描述系统中的实体及其关联。如图 7-1 所示的是每个气象站所在环境中的系统都包含气象信息系统、机载卫星系统和控制系统。连线上的基数关系说明环境中有一个控制系统, 但有多气象站, 以及一个卫星和一个通用气象信息系统。

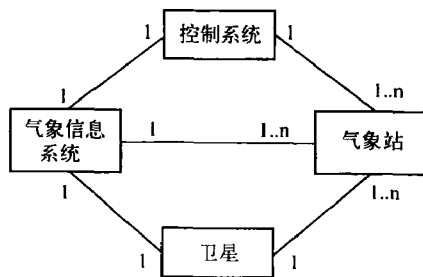


图 7-1 气象站系统上下文

当要对系统和环境间的交互建模时, 需要使用无需太多细节的抽象方法。实现这一点的一种方法是采用用例模型。在第 4 章和第 5 章已经讨论过, 每个用例代表一个与系统的交互过程。每个可能的交互用一个带有名字的椭圆来表示, 外部参与交互的实体用一个人形图形来表示。

气象站系统的用例模型如图 7-2 所示, 图中表明气象站通过与气象信息系统的交互来报告气象数据以及气象站硬件的状态。其余的交互则是与发出具体气象站指令的控制系统进行的。第 5 章解释过, 一个人形图形在 UML 中表示其他系统以及人类用户。

这些用例中的每一个都应用结构化的自然语言来描述, 这有助于设计者识别系统中的对象和理解系统的意图。这里使用一种标准格式以清楚地描述要交换什么信息、交互是如何被启动的等。图 7-3 给出了图 7-2 中报告天气用例的描述。其余一些用例的例子可以在网上找到。

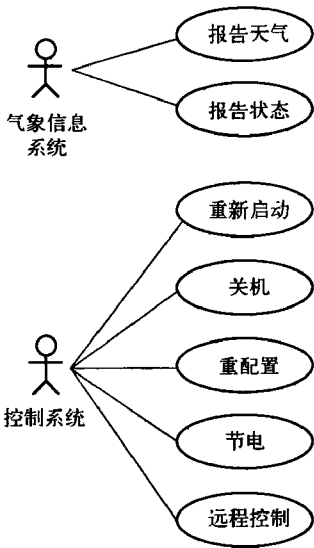


图 7-2 气象站系统用例

系统	气象站系统
用例	报告天气
参与者	气象信息系统，气象站
数据	气象站向气象信息系统发送一段时间内仪器采集的气象数据汇总。数据包括地表和空气的最大、最小和平均温度，大气压强的最大、最小和平均值，风速的最大、最小和平均值，总降水量，以及每五分钟间隔采样的风向统计
激励	气象信息系统与气象站建立卫星通信链路，并要求传输数据
响应	向气象信息系统发送汇总的数据
备注	通常要求气象站每小时进行一次报告，但不同气象站的频率会有所不同，也可能会在将来被修改

图 7-3 用例描述——报告天气

7.1.2 体系结构的设计

一旦完成软件系统与系统所在环境间交互的定义，就可以将这些信息作为系统体系结构设计的基础。当然，还需要结合有关体系结构设计的一般性知识和具体的领域知识。首先识别出组成系统的主要组件以及它们之间的交互，然后运用一种体系结构模式，如分层模式或客户机-服务器模式，来组织这些组件。但这些在此阶段并不是必要的。

气象站软件的高层体系结构设计如图 7-4 所示。气象站由独立的子系统组成，它们之间通过共有的设备进行广播消息来实现通信，如图 7-4 中的通信链路所示。每一个子系统都监听这个基础设备，以获取发送给它们的消息。除了第 6 章介绍的体系结构风格，这也是一种常见的类型。

举例来说，当通信子系统接收到控制指令，如关机，这条指令同时也被所有其他子系统接收，因此各子系统也正确地进行了关机操作。这种体系结构的主要优点在于它易于支持不同配置的子系统，因为消息的发送方无需特别指定消息的接收子系统。

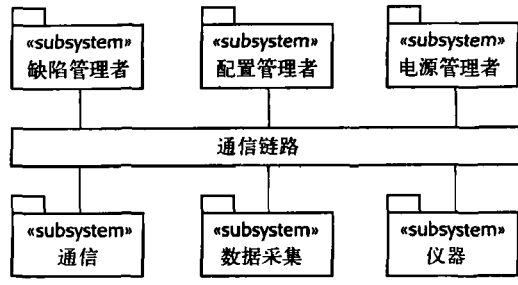


图 7-4 气象站系统的高层体系结构

如图 7-5 所示的是数据采集子系统的体系结构，已经包含在图 7-4 中。Transmitter 和 Receiver 对象与通信管理有关，WeatherData 对象封装仪器采集的信息并将其传输给气象信息系统。这种结构属于生产者 - 消费者模式，将在第 20 章中讨论。

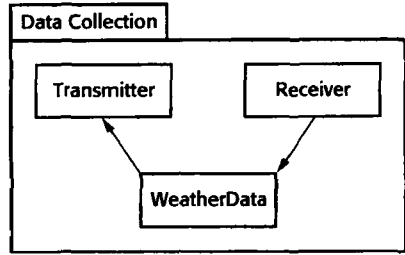


图 7-5 数据采集系统体系结构

7.1.3 对象类识别

在设计过程的这个阶段之前，我们已经对正在设计的系统中的主要对象有了一些认识。随着对设计的理解逐步加深，我们会精炼这些关于系统对象的想法。用例描述有助于识别系统中的对象及其操作。从报告天气用例的描述可以看出，显然需要有对象来代表采集数据的仪器，同时也会有一个对象代表汇总好的气象数据。通常还需要一个高层系统对象或若干用来封装用例中所定义的系统交互的对象。分析出这些对象后，我们就可以开始识别系统中的对象类了。

关于该如何识别面向对象系统中的对象类有各种不同的建议：

1. 对要构造的系统的自然语言描述做文法分析。对象和属性是名词，操作或服务是动词 (Abbott, 1983)。
2. 使用应用领域中的真实实体 (例如，飞机)、职务 (例如，管理者或医生)、事件 (例如，请求)、交互 (例如，会议)、位置 (例如，办公室)、机构单元 (例如，公司) 等 (Coad 和 Yourdon, 1990; Shlaer 和 Mellor, 1988; Wirfs - Brock 等, 1990)。
3. 使用基于脚本的分析识别出系统使用的各个脚本，并依次对其进行分析。在每个脚本分析过程中，分析人员要识别出需要的对象、属性和操作 (Beck 和 Cunningham, 1989)。

在实际过程中，必需综合使用许多知识来发现对象类。先是从非形式化的系统描述中识别出对象类、属性和操作，之后再使用应用域知识和脚本分析来细化和扩展这些对象。这些信息可能来自需求文档，也可能来自与用户的讨论或是对现存系统的分析。

在野外气象站系统中，对象识别基于系统中的有形硬件。受篇幅所限，没有在此列举出全部系统对象，仅在图 7-6 中列出 5 个对象类。Ground Thermometer (地表气温计)、Anemometer (风速计) 和 Barometer (气压计) 对象属于应用域对象，从系统描述和场景 (用例) 描述中识别出

了 WeatherStation 和 WeatherData 对象：

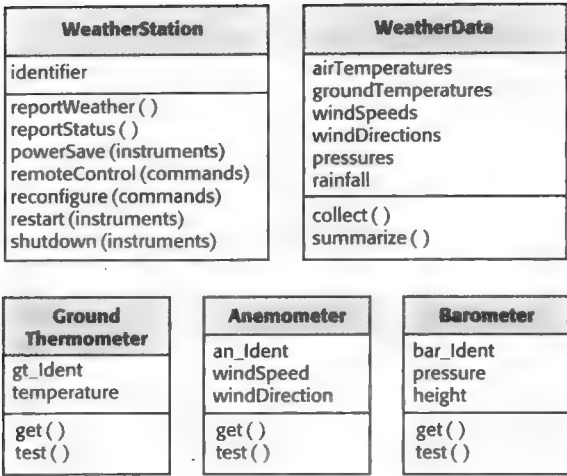


图 7-6 气象站系统中的对象

1. Weatherstation 对象类提供气象站与其环境之间的基本接口。它的操作反映了在图 7-3 中给出的交互。在这一情形下，使用一个单一对象类来封装所有这些交互，而在其他的设计中，使用多个类提供系统接口可能更合适。

2. WeatherData 对象类负责处理报告天气指令，从气象站仪器向气象信息系统发送汇总的数据。

3. Ground Thermometer、Anemometer 和 Barometer 对象类是与系统中仪器直接相关的对象类。它们反映了系统中的有形硬件实体，操作就是对这些硬件的控制。这些对象以某些特定频率自动采集数据并在本地进行存储。这些数据在得到请求时传输给 WeatherData 对象。

运用应用域知识可以识别出其他对象、属性和服务。我们知道气象站通常位于偏远的地区，而各种仪器有时会出现故障，仪器故障应该自动报告。这就意味着需要一些属性和操作来检查仪器是否在正常运转。由于有很多远程气象站，因此每个气象站需要有自己的标识符（ID）。

在设计过程的这个阶段，需要专注于对象本身，而不去想它们的实现方法。一旦识别出对象，就要进行对象设计的精练，在对象间寻找共同点并为系统设计继承层次。例如，定义一个 Instrument 超类，包含所有仪器的共同点，例如一个标识符、get 操作和 test 操作。也可以向超类添加新属性和操作，例如维护数据采集频率的属性。

7.1.4 设计模型

设计或系统模型，如第 5 章中所讨论的，表示系统中包含的对象或对象类，同时也描述了实体间的关联与关系。这些模型是系统需求和系统实现之间的桥梁。首先是对模型进行抽象，不要让琐碎的细节信息扰乱了我们对系统需求的正确理解，同时，设计模型也必须为程序员提供充分的细节以便做出实现决策。

一般地，为了避免这类冲突，可在不同细节层次开发不同的模型。当需求工程人员、设计者和程序员之间联系紧密时，可能只需要一个抽象模型就可以了，详细的设计决策可以放到系统实现时再做，同时用非正式的讨论形式解决问题。当系统描述者、设计者和程序员之间是间接联系的（举例来说，系统由机构的一个部门设计而由其他部门来实现），就需要更详细的模型了。

因此，设计过程中的一个重要步骤就是要决定需要什么样的设计模型和设计模型的细节层

次。这也依赖于所开发的系统类型。设计一个顺序处理数据的系统不同于设计一个实时嵌入式系统，因此需要不同的设计模型。UML 支持 13 种不同的模型，但如第 5 章中所述，很少会用到全部的类型。减少模型使用数量将降低设计的成本和完成设计过程所需要的时间。

用 UML 进行设计通常需要如下两类设计模型：

- 1. 结构模型，通过系统对象类及其之间的关系来描述系统的静态结构。在这一阶段需要记录的重要关系有泛化（继承）关系、使用/被使用关系和组成关系。
- 2. 动态模型，描述系统的动态结构和系统对象之间的交互。需要记录的交互包括由对象发出的服务请求序列以及由这些对象交互触发的状态变化。

在设计过程的早期阶段，有 3 个模型特别有助于为用例和体系结构模型增加细节：

- 1. 子系统模型，给出对象的逻辑分组即子系统，子系统之间存在着有机联系。这些模型用类图的形式来表示，每个子系统作为一个包，里面封装了一些对象。子系统模型是静态（结构）模型。
- 2. 时序模型说明对象交互的序列。使用 UML 时序图或协作图来表示。时序模型是动态模型。
- 3. 状态机模型说明单个对象如何响应事件来改变它们的状态。它们使用 UML 的状态图来表示。状态机模型是动态模型。

子系统模型是一个有用的静态模型，它说明如何将设计组织成逻辑上相关的对象组。图 7-4 中用这种模型来说明在气象制图系统中的子系统。和子系统模型一样，我们也可以设计一些详细的对象模型，描述系统中的对象及其关系（继承、泛化、聚合等）。但这样会有过量建模的趋势，设计过程不能做太详细的关于实现的决定，这些问题应该留给系统程序员。

时序模型是动态模型，描述每种交互模式下发生的对象交互序列。在记录一个设计的时候，我们应该为每一个重要的交互制作一个时序模型。如果已经有了用例模型了，那么我们就应该为所找出的每一个用例制作一个时序模型。

图 7-7 是一个时序模型的例子，描述了 UML 的时序图。这张图表示一个外部系统向气象站请求汇总数据时发生的交互序列，要从上到下阅读时序图。

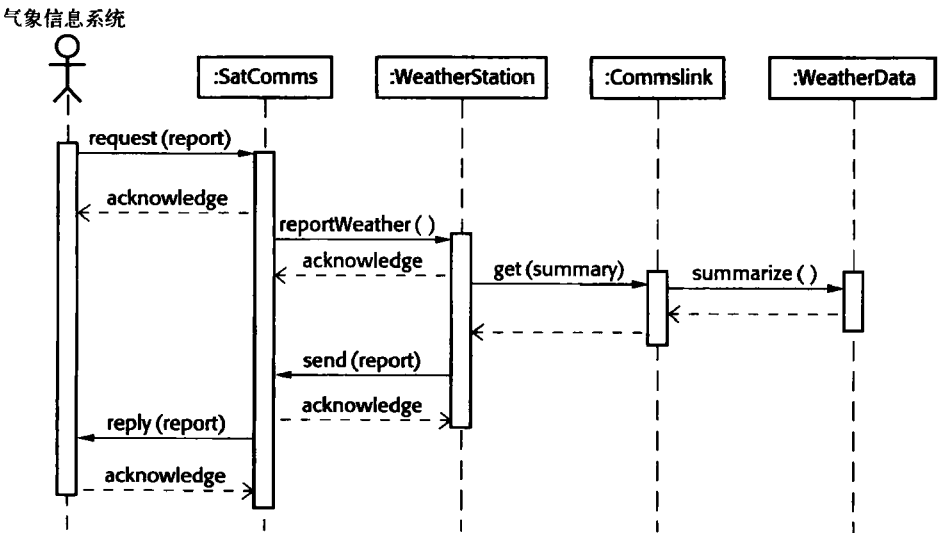


图 7-7 描述数据采集的时序图

1. SatComms 对象接收从气象信息系统发来的天气报告请求,并发出收到回复。已发消息中的实线箭头表明外部系统并不等待回复,继续进行其他处理。

2. SatComms 通过卫星链接向 WeatherStation 发送消息,创建一个采集气象数据的汇总。实线箭头同样表明 SatComms 不等待回复。

3. WeatherStation 向 CommsLink 对象发送包含数据汇总的消息。这里的虚线箭头表明 WeatherStation 对象类的实例会等待回复。

4. Commslink 调用 WeatherData 对象中的 summarize 方法并等待回复。

5. 计算出气象数据汇总并通过 Commslink 对象返回给 WeatherStation。

6. WeatherStation 调用 SatComms 对象,通过卫星通信系统将汇总数据传输给气象信息系统。

SatComms 和 WeatherStation 对象会被实现为并发进程,二者的执行过程可以暂停和恢复。SatComms 对象的实例监听来自外部系统的消息,对这些消息进行解码,并初始化气象站系统的操作。

时序图用于建模一组对象的相关行为,但你也会想满足每个对象或子系统是如何对消息和事件做出响应的。我们可以使用状态机模型刻画。状态机描述对象实例是如何依据新接收的消息改变自身状态的。UML 包含状态图,最初是由 Harel (1987) 创建来描述状态机模型的。

图 7-8 是气象站系统的状态图,它描述了气象站对象是如何对请求的各种服务进行响应的。你可以如下所示去读此图:

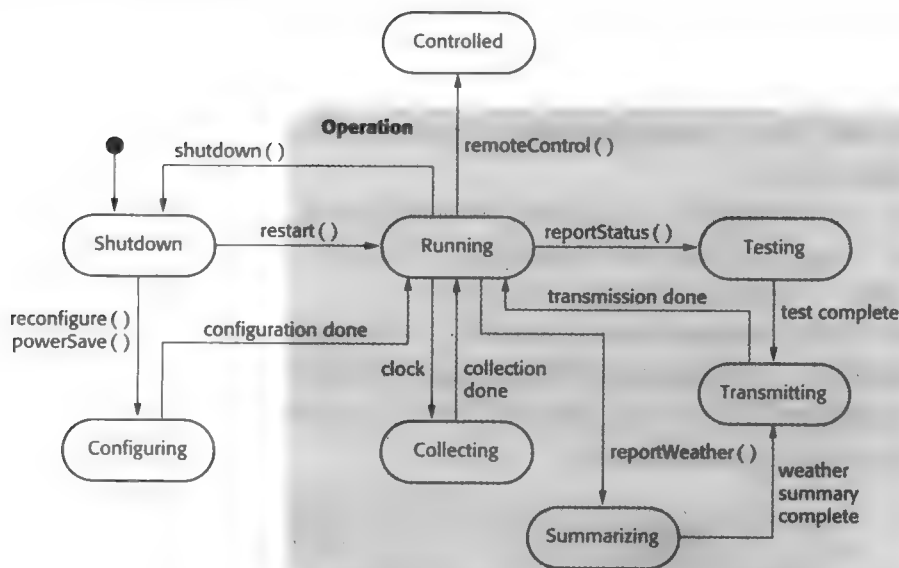


图 7-8 气象站系统状态图

1. 如果对象状态是 Shutdown,则它可以响应 restart()、reconfigure()或 powerSave()消息。带有黑点、未标记的箭头指示 Shutdown 状态是初始状态。restart()消息会引发恢复正常的操作。powerSave()和 reconfigure()消息都会使系统转移到重新配置自身的状态。状态图表明只有在系统已经关机的情况下才允许重配置。

2. 在 Running 状态中,系统期待进一步的消息。如果接收到 shutdown()消息,则对象返回到 Shutdown 状态。

3. 如果接收到一个 reportweather()消息,系统就转移到 Summarizing 状态。当汇总数据完毕时,就转移到 Transmitting 状态,信息传输到远程系统,然后回到 Running 状态。

4. 如果接收到一个 reportStatus()消息,系统转移到 Testing 状态,并在回到 Running 状态前

转移到 Transmitting 状态。

5. 如果收到来自时钟的一个信号, 系统就转移到 Collecting 状态, 开始采集来自仪器的数据。每个仪器被指示依次通过相关的传感器采集它们的数据。

6. 如果接收到 remoteControl() 消息, 系统转移到 Controlled 状态, 此时系统会对远程控制室的另一组消息进行响应, 这一部分没有在图中描述。

状态图是一种很有用的系统或对象操作的高层模型。并不是所有的系统对象都需要状态图。很多系统对象相对简单, 使用状态图反而给设计增加了不必要的细节。

7.1.5 接口描述

对设计中不同组件之间的接口的描述是设计过程的一个重要部分。我们需要详细给出接口描述, 以便对象和子系统能并行地设计。一旦接口已经定义清楚, 其他对象的开发人员就可以假设那个接口已经实现了。

接口设计是关于定义一个或一组对象的详细接口信息。也就是定义对象或对象组所提供的服务的标记和语义。接口可以通过 UML 中与类图相似的方法定义, 只是没有了属性的部分, 同时在类名一栏注明 << interface >>。接口的语义可以用对象约束语言 (OCL) 定义, 这将在第 17 章中解释, 同时给出另一种用 UML 描述接口的方法。

接口设计中不能含有数据表示的细节, 因为接口描述中没有属性的定义, 但需要包含能够访问和更新这些数据的操作。由于数据表示是隐藏的, 因此它可以方便地进行修改而不影响到使用它的对象。这样的设计更容易维护。例如, 一个表示栈的数组可以被更改为列表而不影响到其他使用栈的对象。相反, 将属性暴露在静态设计模型中往往是有意义的, 因为这通常是描述对象本质特性最简洁的方法。

对象和接口间不是简单的一对一的关系, 相同的对象会有若干接口, 每一个都是它所提供的方法上的一个视点。这在 Java 中是直接支持的, 在那里接口是独立于对象声明的, 对象“实现”接口。同样, 一组对象可以通过同一个接口访问。

如图 7-9 所示为气象站系统中定义的两个接口。左边的接口是报告接口, 定义用于生成天气和状态报告的操作名。这些直接映射到 WeatherStation 对象中的操作。远程控制接口提供 4 个操作, 每一个分别映射到 WeatherStation 对象中的一个方法。在此情形中, 每个操作都编码成 remoteControl 方法所关联的一个命令字符串。

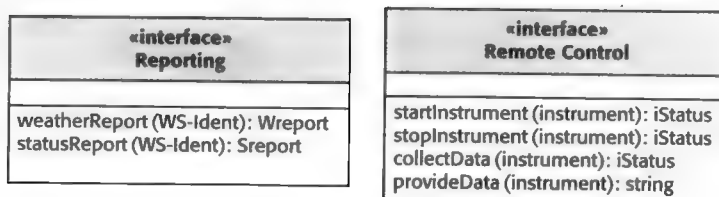


图 7-9 气象站系统接口

7.2 设计模式

设计模式起源于由 Christopher Alexander 提出的概念 (Alexander 等, 1977), 他提出设计的某些模式是相同的, 而且很合意且很有效。所谓“模式”是对问题和解决方案的基本内容的描述, 所以该解决方案可以在不同的设置下复用。模式不是一个详细描述。实际上, 可以把它当做累积的智慧和经验的描述。它是对一般问题的一个经过多次提炼的解决方案。在这一点上, 模式

可以用在分析期间，用以开发系统模型，同时也可以用在设计过程当中。

在此，我们引用 Hillside 研究组网站 (<http://hillside.net>) 上的一段话，这是一个有关模式信息维护的网站，此段话概述了模式在复用中的作用：

模式和模式语言是对最好的经验和好的设计的描述，以一种可以让其他人复用此经验的方式来捕获经验。

模式对面向对象的软件设计有着巨大的影响。除了作为常见问题的经受检验的解决方案外，模式已经成为了谈论设计必用词汇。因此，可以用对所用模式的描述来解释我们的设计。尤其是对“四人帮”（GoF）在他们的关于模式的书（Gamma 等，1995）中最初提到的那些著名的设计模式就更是如此。其他一些特别重要的模式的描述可以从西门子公司、欧洲大型技术公司的作者编写的从书中看到（Buschmann 等，1996；Buschmann 等，2007a；Buschmann 等，2007b；Kircher 和 Jain，2004；Schmidt 等，2000）。

设计模式通常和面向对象设计相关。现有的模式通常依赖于对象的特性，例如继承或多态，以提供通用性。但是，模式中封装经验的一般原则对于任何的软件设计都适用。因此，可以给 COTS 系统一个配置模式。模式是对其他设计师的知识和经验的一种复用。

“四人帮”在其书中定义了设计模式的 4 种主要元素：

1. 名字，是模式的一个有意义的指代。
2. 问题域的描述，解释什么时候模式可以应用。
3. 对部分设计的解决方案描述，描述设计方案的各个部分及其之间的关系和职责。它不是一个具体的设计描述，而是一个设计方案的模板，可以用不同的方式实例化。它通常表达为图形的方式，示意出解决方案中对象及对象类间的关系。
4. 结果陈述，说明应用该模式的结果和副作用。用来帮助设计者了解是否一个模式在特定环境条件下是有效的。

Gamma 和他的合著者将问题描述分解成动机（描述为什么模式是有用的）和适用性（描述模式可以使用的条件）。在解决方案的描述中，将描述模式的结构、参与者、合作和实现。

为说明模式描述，使用观察者模式，如图 7-10 所示，该模式是从 Gamma 的书中摘录的

<p>模式名字：观察者</p> <p>描述：将对象状态的显示从对象本身分离出来，允许存在多种显示。当对象状态变化的时候，自动地通知所有的显示而且让它们得到升级以反映这个变化。</p> <p>问题描述：在许多情况下，提供对对象状态信息的多重显示（如图形显示和图表显示）是非常必要的。不是所有的显示方式在对信息进行定义时就知道的。所有的替代表示法都应该支持交互，当状态改变时，所有的显示必须更新。</p> <p>在需要对对象状态信息有多重显示格式的所有情形下都可以使用该模式。也可以用于对象无需维护显示信息的情形。</p> <p>解决方案描述：包含两个抽象对象，即主体和观察者，还有两个具体对象，即 ConcreteSubject 和 ConcreteObserver，它们继承相关的抽象对象的属性。抽象对象包含可以应用到所有情形的通用操作。要显示的状态由 ConcreteSubject 来维护，它继承来自 Subject 对象的操作，允许它添加和删除 Observer 对象（每个观察者对应一个显示），并在状态发生改变时发出通知。</p> <p>对象 ConcreteObserver 维护 ConcreteSubject 的状态的一个拷贝，并实现观察者的 Update() 接口，该接口允许这些拷贝保持同步。ConcreteObserver 对象自动地显示它的状态，并在状态更新的任何时刻反映变化。</p> <p>这个模式的 UML 模型如图 7-12 所示。</p> <p>结果：主体只知道抽象 Observer，并不知道具体类的细节。因此在这些对象之间有最小的耦合。因为知识的缺乏，对提高显示性能的优化是不现实的。对主体的变更可能引发对要生成的观察者的一系列关联的变更，有些是没有必要的。</p>
--

图 7-10 观察者模式的描述

(Gamma 等, 1995)。这里使用 4 个必要的描述元素, 并附上有关模式能做什么的简短说明。该模式可以使用在需要有不同的对象状态表现的情形下。这个模式将必须显示的对象和不同的显示方式相分离。如图 7-11 所示, 该图示意了相同数据集合的两个图形表示。

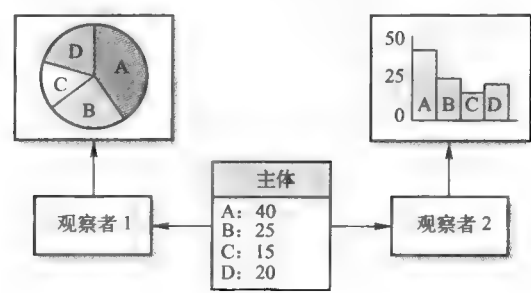


图 7-11 多重显示

图形表示法通常用来说明模式中所用的对象类及其之间的关系。图形表示是对模式描述的补充并增加了解决方案描述的细节。图 7-12 是观察者模式的 UML 形式的描述。

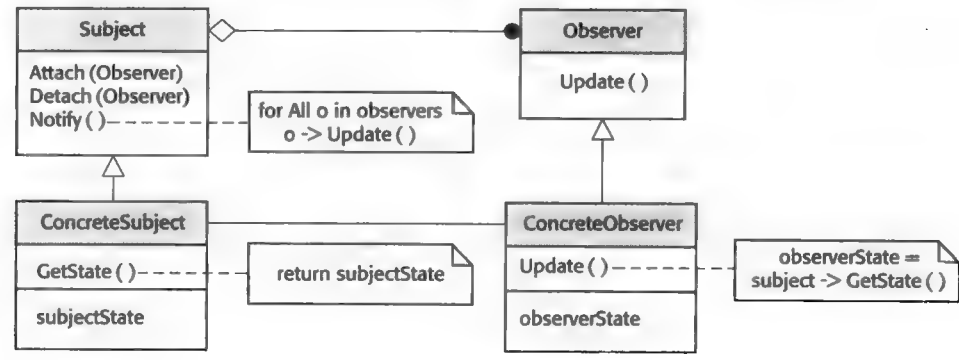


图 7-12 观察者模式的 UML 模型

要在自己的设计中使用模式, 我们要相信我们所面临的任何设计问题都可以用某种相关模式来解决。“四人帮”的模式书中所列出的关于此类问题的例子包括:

- 1. 告知一些对象某个其他对象的状态已经发生改变 (观察者模式)。
- 2. 将接口整理到一些相关对象中, 这些对象通常是增量式开发出来的 (外观模式)。
- 3. 提供一个访问集合元素的标准方法, 无论该集合是如何实现的 (重复器模式)。
- 4. 允许在运行时对已有类进行功能扩展 (装饰器模式)。

模式支持高层的概念复用。尝试复用可执行的组件时, 我们会不可避免地受到该组件的实现人员做出的详细的设计决定的限制。这些限制包含很多内容, 从用于实现组件的算法到组件接口中的对象和类型。当这些设计决定与我们的特殊需求相冲突时, 就不能复用组件, 或者导致系统低效。使用一个模式时, 我们可以复用该模式的思想, 但需要根据我们所开发的系统来调整实现方式。

开始设计一个系统时, 很难确定是否需要一个特定的模式。因此, 在设计过程中使用模式通常包括: 进行设计, 体验问题, 然后找到一个可用的模式。我们当然有可能在原始模式书中所记载的 23 种通用模式中找到答案, 但有时候我们遇到的问题是一个不同的问题, 那么你会发现, 在已提出的数百个模式中寻找一个合适的解决方案不是件容易的事。

模式是一个绝妙的想法, 但需要知道每个模式适用的情况, 因此要在软件设计中不断积累

经验才能学会有效地运用模式。缺乏经验的程序员，即便是阅读过介绍模式的书籍，也会苦于确定是应该复用一个模式，还是应该开发一个专门的解决方案。

7.3 实现问题

软件工程包括从系统的初始需求到已部署系统的维护和管理的所有软件开发活动。其中至关重要的阶段自然是软件实现，在此阶段实现软件的可执行版本。实现包括用高级或底层编程语言开发程序，或者调整现成系统以满足一个机构的特殊需求。

假定本书的大部分读者能理解编程的原则并有一些编程经验。由于本章意在提供一个语言无关的方法，没有侧重于良好的编程习惯问题，因为这需要用特定语言举例说明。取而代之的是介绍实现中一些编程教科书里通常没有介绍过的、而在软件工程中又尤为重要的方面。主要包括：

1. **复用** 多数现代软件都是对现有组件或系统的复用。开发一个软件时，应该尽可能地多用现有代码。
2. **配置管理** 开发过程中，每个生成的软件组件都会有很多不同版本，如果没有很好地在配置管理系统中追踪这些版本，有可能在系统中使用错误版本的组件。
3. **宿主机 - 目标机开发** 软件产品通常不会在与软件开发环境相同的计算机上运行。更多的是开发时使用一台计算机（宿主机），运行时使用另一台计算机（目标机）。宿主机和目标机的系统也有可能是一样的类型，但是更经常是完全不同的环境。

7.3.1 复用

从20世纪60年代到90年代，大部分新软件都是从头开始开发的，用高级编程语言写全部的代码。唯一有意义的复用或软件是对函数和编程语言库中对象的复用。但是，预算和进度的压力使得这种方法越来越不实用，特别是商业的和基于互联网的系统。因此，出现了一种基于复用现有软件的开发方式，并逐步用于商业系统、科学软件，以及嵌入式系统工程。

软件复用可以运用在不同层次上：

1. **抽象层** 这一层中并不是直接复用软件，而是运用软件设计中的成功抽象。抽象知识复用的代表方式是设计模式和体系结构模式（见第6章）。
2. **对象层** 在这一层可直接复用库中的对象，代替自己编写代码。实现这类复用时，必须找到一个合适的库，分析对象和方法是否提供所需的功能。例如，如果需要在Java程序中处理邮件消息，可以使用JavaMail库中的对象和方法。
3. **组件层** 组件是一组通过相互合作实现相关功能和服务的对象及对象类集合，通常需要添加自己的代码对组件进行调整和扩展。利用框架搭建用户接口就是组件层复用的一种，该框架包含一组通用对象类来实现事件处理、显示管理等，只需增加显示数据的连接及定义显示细节的代码，如显示布局和颜色等。
4. **系统层** 在这一层，我们复用整个应用系统。通常涉及这些系统的相关配置。通过添加和修改代码（如果复用的是软件的生产线）或者通过使用系统自身的配置界面。目前多数商业系统都是通过调整和复用COTS（商业现货系统）系统来完成的。有时这种方法涉及对数个系统的复用和集成来创建一个新系统。

通过复用现有软件，可以更快地开发新系统，同时降低开发风险和花费。由于复用的软件已经在其他的应用程序中进行过测试，因此比新软件的可靠性更高。但是，复用也有相应的花费：

1. **寻找可复用的软件并评估其是否符合需求的时间花费。**我们必须通过测试软件来确定该软件可以在现有环境下工作，特别是当此环境异于该软件的开发环境时。

2. 找到合适的可复用软件后, 还要有购买此软件的花费。大型现成系统的花费相应的也会较高。

3. 调整和配置可复用软件组件或系统使其满足待开发系统需求的花费。

4. 集成各个可复用软件(如果使用的是不同来源的软件)及新代码的花费。由于提供商会对各自软件的复用方式做出相冲突的假设, 因此集成不同提供商提供的可复用软件会是很困难的且花费也会很高。

如何复用现有知识和软件是开始软件开发工程时需要考虑的首要问题。在设计软件细节之前就要考虑复用的可能性, 这样就可以根据复用软件的优势调整设计。第2章讨论过, 在面向复用的开发过程中, 需要搜索可复用的元素并修改自身需求和设计将它们各自的优势发挥到最佳程度。

对于多数应用系统来说, 软件工程意味着软件复用。因此, 在本书的软件技术部分使用数章的篇幅(第16、17和19章)对此进行了论述。

7.3.2 配置管理

在软件开发中, 随时都发生着变更, 因此变更管理是必需的。当一个团队开发软件时, 需要确保团队成员彼此之间不妨碍各自的工作。也就是说, 如果将同一个组件分配给两个人时, 那么他们协调他们做出的变更。否则, 一个程序员做出修改后会覆盖了另一个人的工作。此外还需要保证每个人都能访问软件组件的最新版本, 否则开发人员会重做已经完成的工作。当新版本出现问题的時候, 要可以回退到系统组件的正常版本。

配置管理指的是管理变更中软件系统的一般过程。配置管理的目标是支持系统集成过程, 使得每个开发人员可以在管理中访问工程代码和文档, 查找变更, 编译连接组件并生成系统。因此, 配置管理包含如下3个基本活动:

1. 版本管理, 对软件组件不同版本的追踪提供支持。版本管理系统包括协调多个程序员开发的机制, 可以防止一个程序员覆盖其他人提交到系统中的代码。

2. 系统集成, 即提供支持帮助开发人员定义在创建每个系统版本时所用的组件版本。这些描述可以用于编译连接需要的组件以自动构建一个系统。

3. 问题追踪, 即提供支持允许用户报告缺陷及其他问题, 并允许开发人员查看谁在修复这些问题, 以及何时完成的修复。

软件配置管理工具支持上述各种活动。这些工具要设计成能在一个综合的变更管理系统(例如 ClearCase)中协同工作, (Bellagio 和 Milligan, 2005)。在集成的配置管理系统中, 版本控制、系统集成和问题追踪工具是设计在一起的。三者共有一套用户界面风格, 通过共享代码容器集成于一体。

除此之外, 独立工具, 即安装在一个集成开发环境中的, 也会使用到。版本管理可以使用版本管理系统, 例如 Subversion (Pilato 等, 2008), 得到支持。Subversion 能够支持多地点、多团队开发。系统集成支持可以内建到编程语言中, 或者依赖于一个独立工具集, 如 GNU 构建系统。这是一个 Unix 下最知名的集成工具。缺陷追踪或问题追踪系统, 比如 Bugzilla, 广泛应用于缺陷和其他问题的报告以及是否得到修正的追踪。

由于变更和配置管理在专业化软件工程中的重要地位, 因此第25章将进行更详细的讨论。

7.3.3 宿主主机 - 目标机开发

多数软件开发是基于宿主主机 - 目标机模型的。软件在一台计算机(宿主主机)上开发, 但在另一台机器(目标机)中运行。一般的, 我们称之为开发平台和运行平台。平台不只是指硬件

设备, 同时还包括安装的操作系统及其他支持软件, 如数据库管理系统, 或者开发平台中的集成开发环境。

有时, 开发和运行平台是相同的, 这样就可以在同一台计算机上开发和测试软件。但一般情况下二者却是不同的, 因此需要将开发好的软件迁移到运行平台进行测试或者在开发环境下安装模拟器。

模拟器通常在开发嵌入式系统时使用, 以模拟一个硬件设备, 如传感器, 以及系统所要部署的环境中的事件。模拟程序可以加快嵌入式系统的开发过程, 因为每个开发人员可以自己搭建运行环境, 不必再下载软件到目标机硬件中。但是, 开发模拟器的花费较高, 因此只有对当前最常见的几种硬件架构的模拟。



UML 部署图

UML 部署图说明软件组件是如何在处理器上进行实际部署的。也就是说, 部署图描述的是系统中的硬件、软件, 以及连接不同组件的中间件。本质上讲, 我们可以将组件图看做定义和记录目标环境的一种方法。

<http://www.SoftwareEngineering-9.com/Web/Deployment/>

如果目标系统安装了中间件或者其他要用的软件, 那就需要用那些软件来测试系统。而鉴于许可证限制, 在开发机器上安装这些软件是不实际的, 即便是开发平台与目标平台相同。这种情况下, 就需要将开发代码传输到运行环境下进行测试。

软件开发平台应该提供一系列工具来支持软件工程过程, 包括:

1. 集成编译器和句法导向的编辑系统, 能够创建、编辑和编译代码。
2. 编程语言调试系统。
3. 图形编辑工具, 例如编辑 UML 模型的工具。
4. 测试工具, 如 JUnit (Massol, 2003), 可以在新版本的程序上自动运行一组测试。
5. 项目支持工具, 能够帮助我们为不同的开发项目组织代码。

除了这些标准化工具外, 开发系统还可能包含一些特殊的工具, 如静态分析器 (见第 15 章)。通常, 团队开发环境还包括一个共有的服务器, 运行着变更和配置管理系统, 或支持需求管理的系统。

软件开发工具集成在一起形成了集成开发环境 (IDE)。IDE 是一系列支持不同方面软件开发的软件工具, 包括一些常用的框架和用户界面。通常, IDE 是针对某个编程语言 (如 Java) 而开发出来的。编程语言 IDE 是专门开发出来的, 或者是对通用 IDE 的具体化, 并配有针对编程语言支持的工具。

通用 IDE 是一种框架, 它可以将软件工具和集成机制整合在一起。这些软件工具为正在开发中的软件提供数据管理功能, 而集成机制可以让这些软件工具在一起协同工作。最知名的通用 IDE 是 Eclipse 环境 (Carlson, 2005)。这个环境是基于插件体系结构的, 所以它可以针对不同的开发语言和应用领域进行专门处理 (Clayberg 和 Rubel, 2006)。因此, 可以在装 Eclipse 之后根据自己的特定需求通过添加插件去裁剪它。例如, 我们可以添加一组插件支持 Java 的网络系统开发, 或者用 C 语言的嵌入式系统工程开发。

作为开发过程的一部分, 我们需要决定如何将开发完毕的软件部署到目标平台上。这一点对于嵌入式系统来说很简单, 因为目标机通常只是一台计算机。但是, 对于分布式系统来说, 我们需要决定组件部署到何种特定平台上。做决定时需要考虑到的问题有:

1. **组件的软硬件需求** 如果一个组件是为特定硬件体系结构而设计的, 或者依赖于一些其

他软件系统,那么显然需要部署在一个提供所需软硬件支持的平台上。

2. **系统可用性需求** 高可用性系统需要组件部署在多个平台上。当一个平台出现故障,还可以用到备用的组件。

3. **组件通信** 如果组件间有高层通信,通常需要将它们部署在同一个平台,或者距离比较近的平台。这样可以减少通信的延迟,也就是从一个组件发出消息到对方接收到消息的时间。

我们可以通过 UML 部署图来记录软硬件部署的决策,以描述软件组件如何分布于不同硬件平台中。

如果需要部署一个嵌入式系统,就需要考虑目标机的特性,例如大小、电源功率、对传感器事件的实时响应需求、执行器的物理特性,以及它的实时操作系统。我们会在第 20 章中讨论嵌入式系统工程。

7.4 开源开发

开源开发是软件开发的一种方法,它是将软件系统的源代码公布出来,并允许志愿开发人员加入到开发过程中来(Raymond, 2001)。这种方法源于自由软件基金会(<https://www.fsf.org>),该组织主张源码不被任何人所有,而对用户开放,可以随意对其进行检查和修改。前提假设是代码是由小部分核心成员控制和开发的,而不是代码的用户。

开源软件利用互联网发展了大量志愿开发人员,由此扩充了这个理念。多数人同时也是代码的用户。至少在原则上,任何开源项目的贡献者都可以报告和修复缺陷、提议新功能和特性。但是,实际上,成功的开源系统仍然依赖于控制这些软件变更的核心开发小组。

最著名的开源产品自然是 Linux 操作系统,该系统被广泛用作服务器系统,并逐渐成为桌面环境的操作系统。其他重要的开源产品还包括 Java、Apache 服务器和 MySQL 数据库管理系统。计算机产业中主要公司,如 IBM 和 Sun,都支持开源运动,并将自己的软件基于开源产品。当然还有成千上万的其他不知名的开源系统和组件。

获得开源软件一般都是很便宜的或是免费的,不需付费就可以下载到开源软件。但是,如果我们想要文档和支持,那么就需要为此付费,而这样的花费也不高。使用开源产品的另一个关键好处是成熟的开源系统可靠性都很高。原因是这些系统都有庞大的用户群,而且比起向开发者报告缺陷然后等待下一发行版本的系统,他们更乐意自己为系统修复缺陷。因此缺陷的发现和修复都比过去要迅速很多。

对于一个涉足开源开发的公司来说,有两个开源问题值得考虑:

1. 待开发的产品是否应该使用开源组件?
2. 软件开发过程是否应该采用开源方法?

这两个问题的答案取决于待开发软件的类型和背景,以及开发团队的经验。

如果是在开发一个用于销售的软件,则上市时间和降低成本是很关键的。如果正在开发的领域内有高质量的开源系统可用,那么可以通过使用这些系统来节约时间和成本。但是,如果正在开发的系统有一系列特殊的机构需求,那么开源组件就不会是一个选择。此时你可能需要将自己的软件与已有的、并与可用的开源系统不相容的系统集成在一起。但是即使是这样,修改开源系统也比重新开发所需功能更快、成本更低。

越来越多的公司采用开源方法进行开发。他们的业务模型不是依赖于软件产品的销售,而是在卖对该产品的支持。他们相信让开源社区参与进来可以使得软件开发变得更经济、更迅捷,且可以为软件创建一个用户团体。但需要重申的是,这只是用于通用的软件产品,而非特定的机构的应用。

很多公司认为采用开源方式会向竞争对手泄露商业机密,因此不情愿采用这种开发模型。

但是，如果我们在一个小公司工作，当我们将我们的软件开放，客户便可以放心，因为即便公司倒闭，他们也可以支持该软件。

公布系统源代码并不意味着更多的社会人士会参与到开发中。多数成功的开源产品都是平台软件，而不是应用系统。很少有开发人员对特定的应用软件感兴趣，因此将一个软件系统开源并不能保证有很多人参与。

开源使用许可

尽管开源开发的基本原则是源代码应该自由使用，但是这并不代表任何人可以随意地对代码做任何事。从法律上讲，代码的开发人员（公司或个体）仍然持有该代码。他们可以在开源软件许可中限制人们如何合法地使用代码。有些开源开发人员认为如果开源组件用于开发新系统，那么该系统也应该开源。另一部分人则不对自己的代码设置这个约束。开发出来的系统可以不开源并进行销售。

多数开源许可都衍生自下述3种通用模型：

1. GNU 通用公共许可（GPL）。也被称做“互惠”许可，也就是指如果我们用 GPL 许可证的开源软件，那么我们必须也将自己的软件开源。

2. GNU 宽松的通用公共许可（LGPL）。这是 GPL 许可的变体，我们可以编写链接到开源代码的组件，而无需公布自己的组件源码。但是如果我们更改了这些许可下的组件，那么就需要开源。

3. Berkley 标准分布（BSD）许可。这是一个非互惠许可，我们不会被强制要求公开对开源代码的任何改动。我们可以在用于商业用途的系统中使用这些代码。如果我们使用的是开源组件，那么我们必须知道代码的原作者。

如果使用开源软件作为软件产品的一部分，那么根据开源许可条款就会被迫将我们的产品变成开源，因此许可问题是很重要的。如果我们打算销售我们的软件，那么我们不希望泄露源码，这意味着我们应该避免使用 GPL 许可的开源软件。

如果我们在创建一个运行在开源平台下的软件，如 Linux，那么版权许可就不是什么问题。然而，一旦我们在软件中包含了开源组件，我们便需要设立进程和数据库来跟踪所使用的组件及其许可状态。Bayersdorfer（2007）建议管理带有开源代码项目的公司应该做到以下几点：

1. 建立一个维护所下载和使用的开源组件信息的系统。我们需要保存每个组件的许可，这些组件在其使用期间是符合其许可规定的。版权许可会发生变化，因此我们需要记住我们所接受的许可的状态。

2. 知道不同类型的许可并在使用组件之前了解组件是如何发放许可的。我们可以决定在一个系统使用某个组件，在另一而系统不使用，因为这些系统的用途是不同的。

3. 知道组件的发展路线。我们需要对开发所用组件的开源项目有所了解，这样才能知道将来该组件会怎样发展。

4. 培养人们的开源概念。只通过规定来保证许可条件得到遵循是不够的，还需要教育开发人员开源和开源许可的相关知识。

5. 设立审计系统。开发人员可能会在严格的期限下违背许可证的规定。可能的话需要一个软件对此进行检测和阻止。

6. 加入开源社区。如果我们依赖于开源产品，那么我们应该参与到社区中以帮助、支持他们的开发。

软件的商业模型在改变，建立一个销售专业化的软件系统的业务逐渐变得困难起来。很多公司更倾向于将他们的软件开源，然后向软件的用户销售他们的支持和咨询业务。这种趋势随

着开源软件的大量使用和越来越多开源软件的出现加速升温。

要点

- 软件设计和实现是相互交织的活动。设计的细节层次取决于待开发系统的类型以及是否使用计划驱动或敏捷开发方法。
- 面向对象的设计过程包括系统体系结构的设计、识别系统中的对象、运用不同的对象模型描述设计，以及文档化组件接口。
- 面向对象的设计过程中会产生不同的模型。这些模型包括静态模型（类模型、泛化模型、关联模型）和动态模型（时序模型、状态机模型）。
- 要对组件的接口进行精确的定义，这样其他的对象才可以使用它们。可以使用 UML 接口模板来定义这些接口。
- 开发软件时，我们总是要考虑复用现存组件的可能性，无论是对组件、服务还是对整个系统的复用。
- 配置管理是管理系统变更的过程。当一个团队的人员协同开发软件时，这个过程是十分必要的。
- 多数软件开发过程属于宿主机 - 目标机开发模式。要用宿主机上的 IDE 开发软件，然后传输到目标机运行。
- 开源开发指公布系统中的源代码，这意味着很多人可以提出更改建议并改善软件。

进一步阅读材料

《Design Patterns: Elements of Reusable Object-oriented Software》 这是第一本软件模式手册，它向众多读者介绍了软件模式概念（E. Gamma, R. Helm, R. Johnson and J. Vlissides, Addison-Wesley, 1995）。

《Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Iterative Development, 3rd edition》 Larman 清晰地阐述了面向对象设计的方法，并讨论了 UML 的使用。这本书很好地介绍了利用模式进行设计的方法（C. Larman, Prentice Hall, 2004）。

《Producing Open Source Software: How to Run a Successful Free Software Project》 这本书是对开源软件的背景、许可问题和实际运行开源开发项目的一本详细指南（K. Fogel, O'Reilly Media Inc., 2008）。

关于软件复用的进一步阅读资料会在第 16 章推荐，配置管理的资料在第 25 章中介绍。

练习

- 7.1 使用图 7-3 中使用的结构化方法定义气象站系统的 Report 状态的用例和 Reconfigure 用例。你需要对此处所需的功能给出合理的假设。
- 7.2 假设 MHC - PMS 要用面向对象的方法开发。画出用例图，至少包括该系统的 6 个用例。
- 7.3 使用 UML 的图形化符号来表示对象类，设计以下对象类并标明它们的属性和操作。根据你自己的经验决定应该与以下对象关联的属性和操作：
 - 电话；
 - 个人计算机用的打印机；
 - 个人立体声系统；
 - 银行账户；
 - 图书馆目录。

- 7.4 以图 7-6 中识别出来的气象站系统对象为出发点, 识别该系统中可能用到的其他对象, 为识别出的对象设计继承机制。
- 7.5 开发气象站系统设计, 要求给出在数据采集子系统和采集数据的仪器之间的交互过程, 使用序列图来描述这个交互。
- 7.6 识别出下列系统中可能有的对象, 给出这些系统的面向对象设计。在设计过程中可以给出合理假设。
- 一个日记和时间管理系统, 希望它支持一组同事的会议时间安排。当一个会议包括多人时, 系统在这些人员的日记中找到共同空闲时间并将会议安排在这个时间。如果没有共同的空闲时间可用, 系统就同用户交互来安排他们的日程以便腾出时间参加这次会议。
 - 准备设立一个全自动加油 (天然气) 站, 司机需要使用信用卡消费, 加油泵与读卡机相连, 卡的验证是通过与信用公司计算机通信完成的, 同时给出油量上限, 司机可以按需加油。当油加完, 加油软管自动回位。司机信用卡账户要被减掉相应数额。信用卡在账户计算完后退还给司机, 如果卡是无效的, 系统拒绝加油, 退还此卡。
- 7.7 假设有一组人员要安排一个会议, 用时序图绘出上面的日记管理系统中对象间的交互。
- 7.8 绘制 UML 的状态图, 描述在小组日记系统或加油站系统中可能发生的状态变化。
- 7.9 举例解释为什么配置管理在团队开发软件产品的过程中十分重要。
- 7.10 一个小公司开发了一个专用产品, 为每个客户进行专门的配置。新客户总是有特殊的需求需要插入到系统中, 并愿意为此付费。这个公司有一个机会参加一个新合同的竞标, 如果成功, 可以将现有的客户数目至少翻倍。新客户也希望参与到对系统的配置活动中。解释为什么在此情况下将公司所拥有的软件变成开源是个好主意。

参考书目

- Abbott, R. (1983). 'Program Design by Informal English Descriptions'. *Comm. ACM*, **26** (11), 882-94.
- Alexander, C., Ishikawa, S. and Silverstein, M. (1977). *A Pattern Language: Towns, Building, Construction*. Oxford: Oxford University Press.
- Bayersdorfer, M. (2007). 'Managing a Project with Open Source Components'. *ACM Interactions*, **14** (6), 33-4.
- Beck, K. and Cunningham, W. (1989). 'A Laboratory for Teaching Object-Oriented Thinking'. *Proc. OOPSLA'89* (Conference on Object-oriented Programming, Systems, Languages and Applications), ACM Press. 1-6.
- Bellagio, D. E. and Milligan, T. J. (2005). *Software Configuration Management Strategies and IBM Rational Clearcase: A Practical Introduction*. Boston: Pearson Education (IBM Press).
- Buschmann, F., Henney, K. and Schmidt, D. C. (2007a). *Pattern-oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. New York: John Wiley & Sons.
- Buschmann, F., Henney, K. and Schmidt, D. C. (2007b). *Pattern-oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. New York: John Wiley & Sons.
- Buschmann, F., Meunier, R., Rohnert, H. and Sommerlad, P. (1996). *Pattern-oriented Software Architecture Volume 1: A System of Patterns*. New York: John Wiley & Sons.
- Carlson, D. (2005). *Eclipse Distilled*. Boston: Addison-Wesley.
- Clayberg, E. and Rubel, D. (2006). *Eclipse: Building Commercial-Quality Plug-Ins*. Boston: Addison Wesley.
- Coad, P. and Yourdon, E. (1990). *Object-oriented Analysis*. Englewood Cliffs, NJ: Prentice Hall.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley.

Harel, D. (1987). 'Statecharts: A Visual Formalism for Complex Systems'. *Sci. Comput. Programming*, **8** (3), 231-74.

Kircher, M. and Jain, P. (2004). *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. New York: John Wiley & Sons.

Massol, V. (2003). *JUnit in Action*. Greenwich, CT: Manning Publications.

Pilato, C., Collins-Sussman, B. and Fitzpatrick, B. (2008). *Version Control with Subversion*. Sebastopol, Calif.: O'Reilly Media Inc.

Raymond, E. S. (2001). *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, Calif.: O'Reilly Media, Inc.

Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F. (2000). *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. New York: John Wiley & Sons.

Shlaer, S. and Mellor, S. (1988). *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, NJ: Yourdon Press.

St. Laurent, A. (2004). *Understanding Open Source and Free Software Licensing*. Sebastopol, Calif.: O'Reilly Media Inc.

Wirfs-Brock, R., Wilkerson, B. and Weiner, L. (1990). *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall.

软件测试

目标

本章的目标是介绍软件测试过程并介绍一系列测试技术。读完本章，你将了解以下内容：

- 了解测试各个阶段，从开发过程中的测试到系统客户的验收测试；
- 介绍了可以帮助用户选择测试案例的技术以发现程序的缺陷。
- 了解测试优先的开发，即在写代码之前设计好测试案例，然后自动地运行这些用例。
- 掌握组件测试、系统测试和发布测试的主要差异，以及了解用户测试过程和技术。

测试的目的是试图说明一个程序可以做我们期望它所做的工作和在投入使用之前发现程序的缺陷。当我们测试软件时，使用人工数据来执行这个程序。我们审查测试运行的结果，找出关于程序的非功能属性的错误、异常或其他信息。

测试过程有两个截然不同的目标：

1. 向开发者和用户展示软件满足了需求。对于定制软件，这意味着对用户和系统需求文档中的每个需求至少要有有一个测试。对于通用软件产品，这意味着要对每个集成在产品发布版本中的所有系统特征以及这些特性的组合进行测试。
2. 为了找出软件中的缺陷和不足，即软件的行为是不正确的、所不希望的或不符合它的描述的。这些是软件缺陷的后果。缺陷测试关心的是找出所有不希望出现的系统的行为根源。例如系统崩溃，与其他系统的不期望的交互，不正确的计算和数据毁坏。

有效性测试可达成第一个目标，即在这一阶段中使用系统所希望的使用方式的一组测试案例来测试系统的表现。缺陷测试可达成第二个目标，即在这一阶段测试案例是设计来暴露系统的缺陷。测试案例可以是故意模糊的并且不需要反映系统通常是怎样使用的。当然，在这两个测试方法之间没有确定的边界。在有效性测试过程中，我们将会发现系统的缺陷；在缺陷测试过程中，一些测试案例将会表明这个程序满足它的要求。

图 8-1 会帮助我们理解有效性测试和缺陷测试之间的差别。我们把被测试系统看做是一个黑盒子。这个系统接收来自输入集合 I 的输入数据，在输出集合 O 中产生输出结果。一些输出结果可能是错误的，这些错误输出在集合 O_e 中，这些结果是由于集合 I_e 中的输入而产生的。在缺陷测试中，优先去发现在集合 I_e 中的输入，因为这会暴露系统的问题。有效性测试需要正确的输入来进行测试，这些输入是来自 I_e 之外的。这些模拟系统产生所期望的正确的结果。

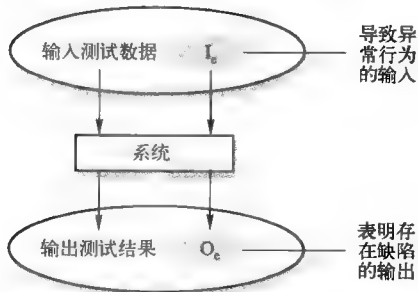


图 8-1 程序测试的输入-输出模型

测试不能够证明软件没有缺陷或者是它在每个环境中都能够按指定的方式执行。我们忽略的某个测试往往能够发现系统其他的一些问题。就像 Edsger Dijkstra, 一个软件开发先驱的富有哲理的话 (Dijkstra 等, 1972):

“测试只能够证明存在错误, 而不能证明它们不存在”。

测试是一个软件更广泛的检验和有效性验证 (V&V) 过程的一部分。检验和有效性验证是不一样的, 尽管它们经常被混淆。Barry Boehm 这位一个软件工程的先驱, 简洁地解释了两者的区别 (Boehm, 1979):

- “检验: 我们在建立正确的产品吗?”
- “有效性验证: 我们建立的产品正确吗?”

检验和有效性验证过程是关于审查被开发的软件是否满足它的描述和符合购买用户所需要的功能。这些审查过程在一有需求时就开始了, 并贯穿整个开发过程。

有效性验证的目的是审查软件满足它的所规定的功能和非功能性需求。然而, 检验是一个更一般的过程。检验的目的是确保软件符合用户的期望。为了证明这个软件的确可以做到用户所期望的, 不简单地是看它是否与需求描述是一致的, 还要看它是否真的是客户所期待地那样去执行的。检验是必要的, 因为正如第 4 章中所讨论的, 需求描述不可能一直反映用户和使用者的真实愿望和系统需求。

检验和有效性验证过程的最终目的是建立信心, 即这个软件系统是“达到目的”的。这意味着这个系统必须足够好。这种对系统的信心水平取决于系统的目的、系统用户的期望和系统当前市场环境:

1. **软件目的** 这个软件越重要, 它的可靠性就越重要。例如, 与一个为展示新产品构思所开发的原型系统相比, 用于控制安全要求极高的系统所需的信心水平要更高一点。
2. **用户期望** 由于他们在问题频出、不可靠的软件上的一些经验, 许多用户对软件的质量抱有很低的期望。当软件出错时, 他们一点也不奇怪。当新的系统建好时, 用户会忍受它的失败, 因为使用的好处超过了故障恢复的成本。在这些情况之下, 我们不必花费更多的时间在测试上。然而, 随着软件的成熟, 用户希望它能更可靠, 所以需要更彻底地测试后期版本。
3. **市场环境** 当一个系统推到市场后, 系统的卖家必须充分考虑竞争的产品和用户愿意付出多少钱来购买这个系统, 以及为实现这一系统所需的时间表。在竞争的环境中, 一个软件公司可能在这个项目完全测试和调试之前就投放到市场中, 因为他们想成为第一个进入市场的公司。如果一个软件产品非常便宜, 用户可能会忍受一个较低的信任度。

如同软件测试、检验和有效性验证过程会涉及软件审查和复查。审查和复查是分析和审查系统需求、设计模型、程序的源代码, 甚至是建议的系统测试, 它们被称作“静态”V&V 技术, 即允许我们不必去执行这个软件而去检验它。图 8-2 说明软件审查和测试在软件过程中的各个阶段对 V&V 的支持。箭头表明在过程中使用这个技术的阶段。

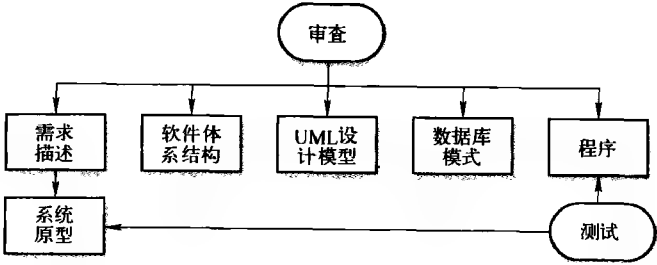


图 8-2 审查和测试

审查主要关心一个系统的源代码，除此之外也涉及任何可读的软件表示，例如它的需求，或是设计模型。当我们审查一个系统时，我们用系统知识、它的应用领域、编程或建模语言来发现它的错误。



测试规划

测试规划是关于对测试过程中所有的活动的进度安排和资源分配。它包括定义测试过程、考虑可用的时间和人员。通常，将要制订一个测试计划，它定义测试什么、预期的测试进度，以及怎样记录测试。对于安全性要求极高的系统，测试计划也可能包括运行在软件上的测试的细节。

<http://www.SoftwareEngineering-9.com/Web/Testing/Planning.html>

相比软件测试，软件审查有 3 个优势：

1. 在测试期间，一个错误可能会掩盖其他错误。当一个错误导致不希望的输出结果时，我们永远不能确信后来的输出异常是由于一个新的错误造成的，还是原来的错误的副作用而导致的。因为审查是一个静态的过程，我们不必关心错误之间的相互作用。所以，一个单个审查阶段可以发现系统中的许多错误。

2. 审查一个系统的不完整版本不需要额外的代价。如果一个程序不完整，我们需要开发特殊测试工具来测试可用的部分。这明显增加了系统的开发成本。

3. 除了搜索程序缺陷，审查也可以考虑一个程序更广泛的质量属性，如符合标准、可移植性和可维护性。我们可以寻找系统低效的维护和更新，不恰当的算法，以及差的编程风格。

程序审查是一很早提出的方法，已经有许多研究和实验证明审查能比程序测试更有效地发现缺陷。Fagan (1986) 报告认为：一个程序中的 60% 的错误可以使用非正式的程序审查来发现。在净室过程中 (Prowell 等, 1999)，据称 90% 以上的程序缺陷可以在程序审查中发现。

然而，审查不能代替软件测试。审查不是很擅长发现以下几种错误。缺陷是由于程序不同部分之间的未预料到的交互而造成的，或因为时序问题所产生的，或因为系统性能问题而导致的。此外，尤其是在小公司或开发组中，把独立的审查小组整合起来是非常困难和昂贵的，因为所有潜在的团队成员也可以是软件开发人员。第 24 章 (质量管理) 将详细讨论复查和审查。自动静态分析，即通过对程序的源文本自动进行分析发现异常情况，将在第 15 章中解释。在本章中，重点放在测试和测试过程上。

图 8-3 是一个“传统”测试过程的抽象模型，用在计划驱动的开发中。测试用例是一个对输入和在特定环境下的期望的输出以及所测试的对象的一个描述。测试数据是为了测试系统而设计的输入。在一些情况下测试数据可以自动产生。但自动测试用例生成是不可能的，因为了解系统应该怎样执行的人必须对期待的测试结果给出定义。然而测试可以自动执行。预期的结果自动地和预测的结果相比较，所以在测试过程中不需要人来查找错误和异常。

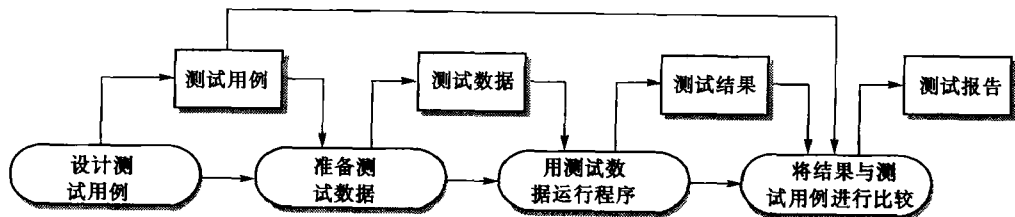


图 8-3 软件测试过程模型

典型地，一个商业软件系统要经过3个阶段的测试：

1. 开发测试，即在开发中进行系统测试来发现故障和缺陷。在测试过程中很可能涉及系统设计师和程序员。
2. 发布测试，即一个测试小组对一个系统的完整版本进行测试，然后发布给用户。发布测试的目的是审查系统是否满足系统信息持有者的要求。
3. 用户测试，即系统的用户或是潜在的用户在他们自己的环境中测试这个系统。对于软件产品来说，“用户”可能是一个内部营销组，它决定该软件是否可以投放市场、发布版本和销售。接收测试是用户测试的一种，即客户正式测试一个系统，以决定是否应该从系统供应处接收，或是否需要进一步开发。

在实践中，测试过程通常既包含手工测试也包含自动测试。在手工测试中，测试人员用一些测试数据来执行程序，然后将结果和预期结果进行比较。他们记录下来并报告给程序开发人员有差异的地方。在自动测试中，测试是通过一个程序去执行的，该测试程序会一遍遍地执行正在开发的系统。这个过程通常比人工测试速度要快，尤其是当它要进行回归测试时——重新运行先前的测试以检测对程序的变更没有引入新的故障。

自动测试的使用在过去的几年里有相当大的提高。然而，测试绝不能是完全的自动测试，因为自动测试只能检测一个程序做了它应该做的事。对于一类以观感好坏为评价依据的系统（例如图形用户界面），或者是对于焦点放在程序没有不希望的负效应的测试来说，自动测试是无能为力的。

8.1 开发测试

开发测试包括系统开发团队所进行的所有测试活动。软件的测试人员通常是开发这个软件的编程人员，虽然并非总是如此。一些系统是由编程人员和测试人员配对（结对）开发的（Cusamano 和 Selby, 1998），每个程序员都有一个相关测试人员，这个测试人员开发测试用例并辅助测试过程。对要求极高的系统，会采用一个更正规的测试过程，由独立的测试人员负责所有阶段的测试。他们负责开发测试用例和维护测试结果的详细记录。



程序调试

程序调试是修改由测试所发现的错误和问题的过程。使用来自程序测试的信息，调试人员用他们对程序语言的知识 and 测试所预期的输出结果来定位和修复程序错误。这个过程经常是由交互式调试工具来支持的，这个工具可以提供程序执行的额外信息。

<http://www.SoftwareEngineering-9.com/Web/Testing/Debugging.html>

在开发过程中，可在3个粒度级别进行测试：

1. 单元测试，即对单独的程序单元或对象类进行测试。单元测试应该着重测试对象或方法的功能。
2. 组件测试，即将多个程序单元整合创建一个合成的组件。组件测试应该着重测试组件的接口。
3. 系统测试，即集成系统中的一些或所有的组件作为一个整体进行测试。系统测试应该着重测试组件的交互。

开发测试主要是一个缺陷测试的过程，即测试的目的是发现系统中的错误。因此，它通常和调试交错执行。调试是定位有问题的代码的过程，然后修改程序来解决这些问题。

8.1.1 单元测试

单元测试是测试程序组件的过程，程序组件例如方法或对象类。单个函数或方法是最简单的组件形式，我们的测试应该用不同的输入参数调用这些程序。可以使用在 8.1.2 节中所讨论的用于测试用例设计的方法来设计函数和方法的测试。

当我们测试对象类时，我们应该设计测试来提供对对象所有特征的覆盖。这意味着我们必须：

- 测试与对象相关的所有操作；
- 设置和检查与对象相关的所有属性；
- 让对象处于所有可能的状态下，就是说模拟所有能改变对象状态的事件。

作为一个例子，考虑在第 7 章中的讨论过的气象站对象的例子，它的接口如图 8-4 所示。它只有一个属性，就是它的标识符。这是个常量，在气象站建立的时候被设定的。因此我们只需要一个测试，来检查它是否已经恰当设置。我们需要为与对象相关的所有方法定义测试用例，例如 reportWeather, reportStatus 等。理想情况下，我们应当孤立地测试这些方法，但是，在某些情况下，某些测试序列是必要的。例如，为了测试关闭气象站仪器的方法，我们必须执行 restart 方法。

WeatherStation
identifier
reportWeather() reportStatus() powerSave(instruments) remoteControl(commands) reconfigure(commands) restart(instruments) shutdown(instruments)

图 8-4 气象站系统对象接口

泛化或继承使得对象类测试更加复杂。我们不能简单地测试定义在这个类中的操作，然后认为它在继承这些操作的子类中能够按照预期地工作。这个继承的操作可能对关于其他的操作和属性做了假设。这些在某些继承了这个操作的子类中可能是无效的。因此我们必须在所有使用了这个操作的上下文中测试此操作。

为了测试气象站的状态，我们使用一个状态模型，例如在第 7 章中图 7-8 所示意的。使用这个模型，我们可以识别出需要测试的状态转换的序列，并指定迫使这些状态转换发生的事件序列。原则上讲，我们应该测试所有可能的状态转换序列，虽然实际上这可能太昂贵。气象站中需要测试的状态序列例子包括：

Shutdown → Running → Shutdown

Configuring → Running → Testing → Transmitting → Running

Running → Collecting → Running → Summarizing → Transmitting → Running

只要可能的话，我们应该使单元测试自动化。在自动化单元测试中，我们应该充分利用测试自动化框架（例如 JUnit）来编写和运行程序测试。单元测试框架提供了一个通用的测试类，我们只需扩展它来形成新的测试案例。然后它们可以运行已经实现的所有测试，然后通常是通过一些图形用户界面来报告测试成功或失败。一个完整的测试套件通常在几秒内运行完毕，因此它可以允许我们在每次对程序进行一点修改后就执行一次。

一个自动化测试有 3 个部分：

1. 准备部分，用测试用例初始化系统，即输入和期望的输出。
2. 调用部分，即调用所要测试的对象或方法。
3. 断言部分，即比较调用的结果和预期的结果是否相同。如果断言取值为真，那么测试成功；如果断言取值为假，那么测试失败。

有时候，我们正在测试的对象与其他对象有依赖关系，而被依赖对象可能还没有写出来，或使用它们会减慢测试过程。例如，如果测试对象调用一个数据库，这可能涉及一个很慢的准备过程，然后才能使用。在这些测试用例中，我们要决定去使用一个模型对象。所谓模型对象就是这

样一种对象，它们与正在使用的外部对象有相同的接口并仿真其功能。因此，一个仿真数据库的模型对象可能只有很少的数据项存在于一个数组中。因此访问它们很快，没有调用数据库和访问磁盘的开销。同样地，模型对象可以用来模拟异常的操作或罕见的事件。例如，如果我们的系统计划在一天固定时刻采取某个动作，我们的模型对象可以很简单地返回这些时间，而不管真实的时钟时间。

8.1.2 选择单元测试案例

测试是非常昂贵和费时的，所以选择有效的单元测试案例是非常重要的。在这种情况下，有效性意味着两件事：

1. 测试案例应该表明，当按照预期的方式使用时，所测试的组件能够像假设的那样去执行。
2. 如果在组件中有缺陷，这些缺陷应该被测试案例发现。

因此我们应该有两种类型的测试案例。第一种应该能够反映一个程序的正常操作，并且应该能显示出组件工作正常。例如，如果我们正在测试一个组件，这个组件创建并初始化一个新病人记录，然后我们的测试案例应该能够表明这个记录存在于数据库中，并且它的字段已经如输入那样设置。另外一种测试用例应该建立在对通常问题的经验基础上。它应该使用非正常的输入来检测是否得到了正常处理，或者是否使得组件崩溃。

这里所讨论的两种策略，可能会有效地帮助我们选择测试案例。包括：

1. 划分测试，即识别具有共同特性和以同样的方法处理的一组数据。我们应该从这些组中选择测试数据。
2. 基于准则测试，即使用测试准则来选择测试案例。这些准则反映了程序员在开发组件时对经常犯的各种错误的经验。

程序的输入数据和输出结果总是落在几个不同且具有共同特征的类中，例如都是正数、负数和菜单选择。程序通常对一个类中的所有成员其行为都是差不多的。也就是说，如果我们测试一个能进行某种计算的而且要求两个正整数作为参数的程序，我们会期望这个程序能够对所有的正整数以相同的方式执行。

由于这些等价的行为，这些类通常叫做等价划分或是域（Bezier, 1990）。测试案例设计的一个系统化的方法就是对系统或组件识别所有输入和输出的划分。所设计的测试案例要使得输入和输出落在这些划分中。划分测试既可以用来设计系统的测试案例也可以用来设计组件的测试案例。

在图 8-5 中，左边大椭圆阴影代表所有可能的要被测试的程序输入。没有阴影的小椭圆代表等价划分。被测试的程序应以同样的方式处理一个输入等价划分中的所有数据。输出的等价划分是具有某些共同特性的程序输出。有时，输入和输出等价划分是 1 对 1 映射关系。然而，并非总是如此。我们可能需要定义一个单独的等价划分，其输入唯一的相同特性是它们产生的输出落在同一个输出划分中。左边椭圆阴影区域代表无效的输入。在右边椭圆的阴影区域代表可能发生的异常（例如，针对无效的输入的响应）。

一旦找出了一个划分集合，我们就可以从每一个这些划分中选择测试案例。一个选择测试案例的好的经验法则是在划分的边界上并靠近划分中间点的地方选择测试案例。其原因是设计者和程序员在开发系统时倾向于考虑输入典型值。我们可以通过选择划分的中间点值来测试这些典型值。边界值通常是非典型的（例如，零可能与其他的非负数的行为是不同的），所以有时被开发者忽视掉。程序经常在运行这些非典型值的时候失败。

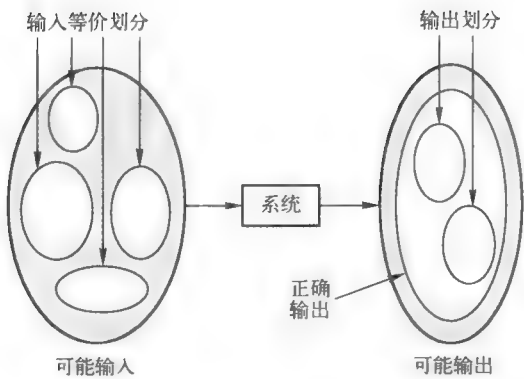


图 8-5 等价划分

我们是通过使用程序描述或用户文档来识别划分，也可以根据经验知识，即预计可能检测出错误的输入值的那些类。例如，假设某个程序描述声明程序接受 4 ~ 8 个输入值，它们都是五位的大于 10 000 的整数。我们使用这个信息来辨别输入划分和可能的测试输入值。如图 8-6 所示。

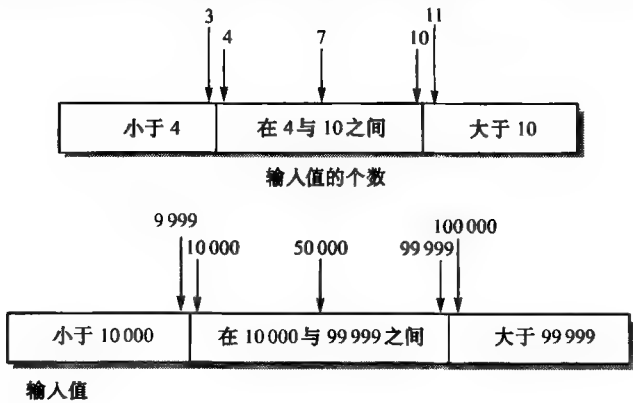


图 8-6 等价分类

当我们使用系统描述来确定等价划分，这被称为黑盒测试。这时，我们不必了解系统是如何工作的。然而，白盒测试对黑盒测试是一个很好的补充，白盒测试时，我们可以看到程序代码来发现可能的测试。例如，我们的代码可能包括处理错误输入的异常。我们可以利用这个知识来确定“异常划分”——相同的异常处理但不同的范围。

等价划分是测试很有效的方法，因为它可以找出程序员在处理边界输入时经常犯的错误。我们可以用测试准则来帮助选择测试案例。测试案例的准则封装关于哪些测试案例对发现错误是非常有效的知识。例如，当我们测试带有序列、数组或者是链表的程序时，有多个准则可以揭露缺陷：

1. 用一个只有单个值的序列来测试程序。程序设计人员往往自然地认为一个序列是由多个值构成的，有时在他们的程序设计中嵌入了这个假设。结果是，当程序接收的是只有单个值的序列时就不能正常工作了。
2. 在不同的测试中使用不同规模的多个序列。这样做减少了有缺陷程序因为输入的某些巧合特性意外地造成正确输出的可能性。
3. 导出一个测试，让第一个、中间一个和最后一个元素得到测试。这个方法能暴露在等价分边界上的一些问题。

路径测试

路径测试是一种测试策略，其目标是要执行组件或程序中的每一条执行路径。如果每一条独立路径都执行过，那么组件中的每条语句就肯定至少执行过一遍。此外，所有的条件语句的真值状态和假值状态都经过了测试。在面向对象的开发过程中，路径测试在测试对象中的方法时常会用到。

<http://www.SoftwareEngineering-9.com/Web/Testing/PathTest.html>

像 Whittaker (Whittaker, 2002) 这些作者把他们的测试经验封装在一组原则中，这些原则增加了缺陷检测成功的概率。下面列出了这些原则的一些例子：

- 选择能够迫使系统产生所有错误信息的输入；
- 设计能够使系统的输入缓冲溢出的输入；
- 重复相同的输入或一系列输入很多次；
- 迫使产生无效的输出生；
- 迫使输出结果太大或太小。

当我们取得一定的测试经验后，我们可以开发自己的选择有效的测试案例的测试准则。本章后面将给出测试准则的很多例子。

8.1.3 组件测试

软件组件通常是由许多彼此交互的对象组合的复合组件。例如，在气象站系统中，再配置组件包括处理再配置各个方面的各对象。我们可以通过它们定义的接口来访问它们的功能。那么测试这些组件时，我们首先关心的就是测试组件的接口行为是否符合它们的描述。我们可以假定组件的单个对象的单元测试已经完成。

图 8-7 说明了组件接口测试的方法。假设通过对组件 A，B，C 的集成构造一个大的组件或子系统。测试案例不是应用于单个组件的，而是应用于由这些组件构成的复合组件的接口。在复合组件中的接口错误是不能通过对单个对象或组件的测试检测到的。因为这是由其中的各对象之间的交互引起的。

在程序组件之间有各种接口类型，由此也就有不同接口的接口错误：

1. **参数接口** 在这些接口中，主要是数据和函数指针，由一个组件传递到另一个组件。对象中的方法有一个参数接口。
2. **共享内存接口** 在这些接口中，有一个被子系统共享的内存块。数据由一个子系统放在其中，被另外的子系统取出。这种类型的接口经常用在嵌入式系统中，即传感器创建数据供其他系统组件检索和处理。
3. **程序接口** 在这些接口中，有子系统封装的一组程序，这些程序可以被其他子系统调用。对象和抽象数据类型有这种接口形式。
4. **消息传递接口** 在这些接口中，子系统通过消息传递来请求其他子系统上的服务。返回的消息会包含这个服务的运行结果。一些面向对象系统有这种接口，同样，客户机 - 服务器系统也有这种接口。

接口错误是复杂系统错误中最常见的形式之一 (Lutz, 1993)。接口错误又可以分为 3 类：

- **接口误用** 调用者组件在调用其他组件时接口使用不当而产生接口错误。这种错误类型特

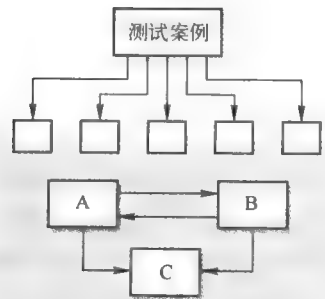


图 8-7 接口测试

别容易发生在参数接口中，往往是因为使用的参数类型不对、传递顺序不对，以及参数数量不对。

■ **接口误解** 调用者组件误解了被调用组件的接口描述而产生接口错误，对被调用组件行为进行了错误的假设。被调用组件没有按照预期的方式动作，从而引起了调用者组件的不可预期的行为。举例来说，一个二分搜索例程用一个未排序的数组参数来调用，这个搜索由此失败。

■ **时序错误** 一般发生在实时系统中，系统使用了共享内存接口或消息传递接口而产生接口错误。数据的生产者和消费者可能有不同的速度。除非在接口设计中特别注意，不然的话，由于生产者来不及更新共享接口信息造成消费者访问了过期的数据。

测试接口缺陷很困难，原因在于接口缺陷往往在不寻常的条件下出现。举例来说，一个对象用固定长度的数据结构实现了一个队列。调用者对象可能认为队列是一个无穷的数据结构，因而未对溢出进行审查便将一个项加入其中。这个条件只能通过设计一个测试用例迫使该队列发生溢出并导致对象行为紊乱，才能使对象的行为容易让测试者觉察到。

更深一层的问题会出现在两个交互的对象或模块都有缺陷的情况下。一个对象当中的缺陷只有当另一个对象行为异常时才能暴露出来。举例来说，一个对象可能调用其他对象上的服务并假定响应是正确的。如果被调用的服务以某种方式出现故障，返回的数值可能虽然有效但却是不正确的。这不能立即检测出来，仅当后来的计算出错时才显现出来。

接口测试的一般准则有：

1. 审查要测试的代码并明确地列出对外部组件的每个调用。设计一组测试集，使得对传给外部组件的参数的值选择紧靠取值范围边缘的那些值。这些特殊取值很有可能暴露接口的不一致性。

2. 当有指针从接口传递时，总用空指针参数来测试接口。

3. 当组件通过程序接口被调用时，设计一些容易引起组件失败的测试。对失败假设的不一致是最平常的描述误解之一。

4. 在消息传递系统中进行强度测试，如在先前一节中讨论的那样。让所设计的测试产生比实际情况多得多的消息。由此时序问题可以显露出来。这是一个有效的揭示时序问题的方法。

5. 当组件间通过共享内存来交互时，可以设计一种测试，使其对激活组件的次序有所改变。这些测试可以暴露程序员对共享数据的生产和消费顺序所进行的隐式的假设。

审查和复查有时比发现接口错误的测试更有效。审查可以专注于组件接口，在审查过程中回答有关假定的接口行为问题。一个强类型语言，例如 Java，允许很多接口错误被编译器捕获。静态分析工具（见第 15 章）能发现很多种接口错误。



增量集成和测试

系统测试包括集成不同的组件，然后测试我们所创建的集成系统。我们应该一直使用增量的方法来集成和测试（例如，我们应该集成一个组件，测试这个系统，集成另一个组件，再测试等）。这意味着如果发生问题，它可能是和最近集成的组件有关。

增量集成和测试是敏捷方法的最基本的方法，例如极限编程和回归测试（参见 8.2 节）在每一次进行增量集成后就会运行。

<http://www.SoftwareEngineering-9.com/Web/Testing/Integration.html>

8.1.4 系统测试

开发中的系统测试包括集成组件来形成一个新版本的系统，然后测试集成后的系统。系统

测试确保组件是可兼容的、能正确地进行交互，以及通过它们的接口在适当的时候传送正确的数据。它显然与组件测试重叠，但有两个重要的区别：

1. 在系统测试中，单独开发的可复用组件和商业现货系统可能会与新开发的组件集成到一起，然后对完整的系统进行测试。
2. 不同小组成员或群组开发的组件可能在这个阶段集成。系统测试是一个集体的过程而不是一个独自的过程。在一些公司中，系统测试可能由一个独立的测试小组执行，没有设计人员和程序员的参与。

当我们集成组件来形成一个新的系统时，我们得到系统的整体行为。这意味着当我们把组件集成到一起的时候系统的某些功能元素才显现出来。这可能是计划中的总体行为，这个必须进行测试。例如，我们可能集成一个认证组件到一个更新信息的组件。然后系统就有了特性以规定只有授权用户方可进行信息更新。然而有时，总体行为并非是计划中的也不是想要的。我们必须开发测试案例来检查这个系统只会做它应该做的。

因此系统测试应该重点测试组成一个系统的组件间和对象间的交互。我们可能还要测试可复用组件或系统以检测当它们和新的组件集成到一起的时候会按照预期的方式工作。交互测试应该发现那些仅能在其他组件使用一个组件的时候才能暴露出来的组件缺陷。交互测试也可以帮助发现组件开发者对其他组件的误解。

因为着重于交互，所以使用基于案例的测试是系统测试一个很有效的方法。典型的，每个案例是通过几个组件或系统对象来实现的。测试一个案例使得系统发生这些交互。如果开发了一个序列图来建模用例实现，我们可以看到在交互中所涉及的对象或组件。

为了说明一点，使用来自野外气象站系统的例子。气象站被要求向远程计算机报告汇总的气象数据。图 7-3 说明了这个用例（请参阅第 7 章）。图 8-8（是图 7-7 的副本）给出的是在气象站中的操作序列，即当它响应请求去收集数据供制图系统使用时的操作。我们可以使用此图来识别将要被测试的操作，并帮助设计测试案例来执行测试。因此发出需要报告的请求将引起下列方法的线程的执行：

```
SatComms:request → WeatherStation:reportWeather → Commslink:Get(summary)
→ WeatherData:summarize
```

序列图可以帮助我们设计我们所需要的特定的测试案例，同时它可以表示什么输入数据是所需要的和产生什么样的输出数据。

1. 对报告请求的输入应该有一个相应的确认，且最终报告会从请求返回。在测试中，我们应该建立一个能用来检测确定报告被正确组织的汇总数据。
2. 向 WeatherStation 索要报告的输入请求导致了一个汇总报告的生成。我们可以孤立地测试它，我们可以创建源数据并审查 WeatherStation 是否能产生正确的汇总，源数据对应于我们已经准备好用于测试 SatComms 的汇总内容。这些源数据也可以用于对 WeatherData 对象的测试。

当然，图 8-8 是简化了的序列图，所以它并没有表现出异常。一个完全的案例/脚本测试必须要考虑到异常情况，保证对象能够正确处理异常。

对于大多数系统来说，很难知道多少系统测试是最低限度的以及何时应该停止测试。无遗漏的测试，即对每一个系统可能的执行序列都进行测试，是不可能的。因此需要建立一个可能的测试案例的子集，根据这个子集进行测试。理想情况下，软件公司应该有选择这个子集的策略。这些策略可能基于一般的测试策略，例如所有的程序语句至少被执行一次的策略等。另外，测试策略也可以基于对系统的使用经验，集中关注对可用的系统的特征。例如：

1. 所有的能从菜单中得到的系统功能都应该被测试到。
2. 可以从同一个菜单中访问的组合功能（例如文本格式化）需要被测试。

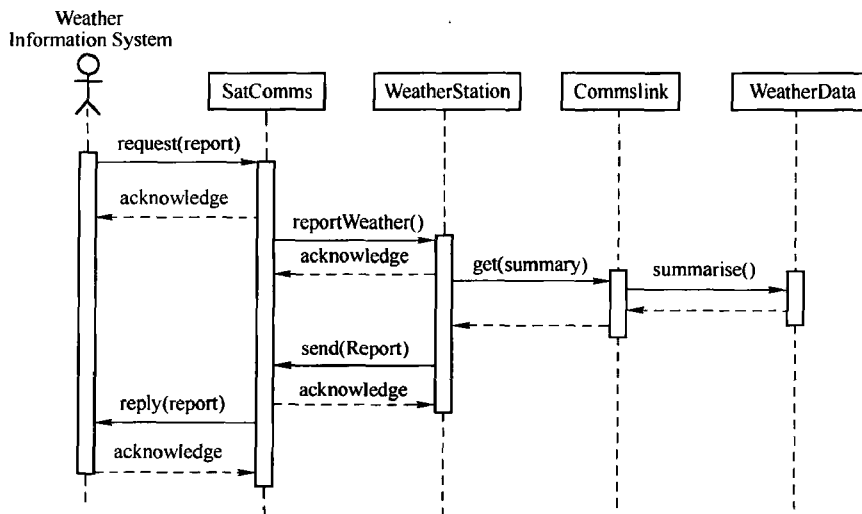


图 8-8 收集气象数据的序列图

3. 在提供用户输入的地方，所有的功能都必须对正确的和不正确的输入进行测试。

来自主要的软件产品例如字处理器或电子表格的经验可以清楚看出，在产品测试当中一般都使用了类似的准则。当孤立地使用软件的某些特性时它们通常能正确地执行。问题出现在当结合使用一些不常用的特性的时候，就像 Whittaker (Whittaker, 2002) 解释的。他给出了一个例子，在一个常用的字处理器中，脚注与多栏输出一起使用时会引起不正确的文本输出。

自动系统测试一般比自动的单元或组件测试困难。自动单元测试依赖一个预测输出，然后将这些预测写成程序。接下来就是将这些预测和结果进行比较。然而，实现一个系统的目的可能是产生一个很大的输出，这个输出或许不能轻易地预测。我们可以做到检查输出并确认它的可信性，而无需提前生成它。

8.2 测试驱动开发

测试驱动开发 (TDD) 是一种程序开发方法，我们交错进行测试和代码开发 (Becker, 2002; Jeffries 和 Melnik, 2007)。随着代码逐步增多，针对代码的测试也会随之增多。直到已开发的代码通过了测试，否则我们不可以开发下一个功能代码。测试驱动开发的引入是作为如极限编程这样的敏捷方法的一部分。然而，它也可用于计划驱动的开发过程中。

测试驱动的基本过程如图 8-9 所示。这个过程步骤如下：

1. 从识别所需要的功能增量开始。这个通常应该比较小，用几行代码就可以实现。
2. 针对此功能编写一个测试并实现为一个自动测试。这意味着测试是可执行的，并将报告它是通过或是失败。
3. 然后运行此测试，以及所有已实现的其他测试。最初，我们并没有实现这个功能，因此这个新的测试将是失败的。这是有意的，因为它可以表明测试添加了一些新东西到测试集中。
4. 然后是实现这个功能，并重新运行这个测试。这可能涉及重构现有的代码来改善它，并添加新的代码到已经存在的代码中。
5. 一旦所有的测试成功，我们可以转去实现下一个功能块。

自动测试环境，例如支持 Java 程序测试的 JUnit 环境 (Massol 和 Husted, 2003)，对于 TDD 来说是必要的。由于代码开发成非常小的增量，所以我们必须能够在每次增加功能或重构程序时都要进行测试。因此，测试是嵌入到一个单独的可以运行测试和调用被测试程序的程序中的。

使用这个方法，可以在几秒内运行数百个单独的测试。

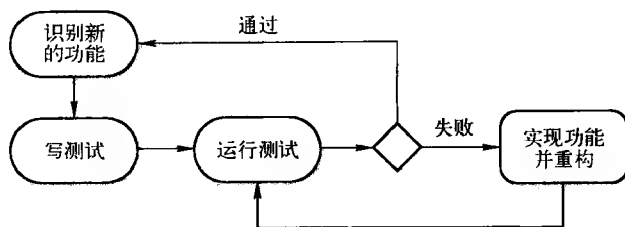


图 8-9 测试驱动的开发

测试驱动的开发的一个强有力的理由是，它可以帮助程序员弄清楚代码段实际上应该做什么。为了写一个测试，我们需要了解目的是什么，因为这样的理解可以使得编写所需要的代码变得很简单。当然，如果我们没有完全的这方面的知识和理解，测试驱动开发也帮不上忙。如果我们没有足够的理解来写测试，我们就无法写出所需要的代码。例如，如果我们的计算涉及除法，我们就应该检查除数没有为 0。如果我们忘记写这样一个测试，检测的代码将不会包括在程序中。

除了对问题有较好的理解外，测试驱动开发还有其他的优势：

1. **代码覆盖** 原则上，我们所写的每个代码片段都至少有一个测试。因此，我们可以确信系统中的所有代码都实际执行过。因为代码边写边进行测试，在开发过程中可以提早发现缺陷。
2. **回归测试** 随着一个程序的开发，一个测试套件也增量式开发出来。我们可以一直运行回归测试来审查程序中的变更没有引起新的错误。
3. **简化调试** 当一个测试失败的时候，问题出在何处是很明显的。新写的代码需要审查和修改。我们不必使用调试工具来对问题定位。使用测试驱动开发的报告显示，在一个测试驱动开发中，几乎没有必要使用自动调试器。
4. **系统文档** 测试本身就表现为一种文档形式，它描述了代码应该做什么。阅读测试可以使得理解代码变得更容易。

测试驱动开发的最重要的优势之一是它减少了回归测试的代价。回归测试包括运行测试集，在对一个系统进行更改后这些测试已成功执行。回归测试就是在系统发生了变更之后重复运行先前曾成功运行过的测试集。回归测试检查这些改变没有引入新的缺陷到系统中来，以及新的代码和已存在的代码能够很好地按预期的方式交互。回归测试是非常昂贵的，当一个系统是人工测试的时候，回归测试有时是不现实的。因为时间成本和人力成本都非常高。在这种情形下，我们必须尝试并选择最相关的测试来重新运行，但这样做很容易丢失重要的测试。

然而，自动化测试，对于测试优先开发来说是至关重要的，它极大地减少回归测试的成本。现有的测试可能会快速地并且成本很小地重新运行。在测试优先开发中，在对系统进行了变更后，所有的已经存在的测试必须成功地运行后，才能添加新的功能。作为一个程序员，我们必须很确信我们所添加的新功能不能为已有代码带来问题。

测试驱动开发在一个新的软件开发中是最经常使用的，其中功能要么是在新的代码中实现的要么是使用经过良好测试过的标准库。如果我们复用大量的代码组件或是遗留系统，我们需要将这些系统作为一个整体来编写测试。测试驱动开发在多线程系统中也是低效的。不同的线程可能会在不同时刻在不同的测试过程中交错运行，所以可能产生不同的结果。

如果我们使用测试驱动开发，我们还需要一个系统测试进程来验证系统。也就是说，检查它符合本系统所有信息持有者的需求。系统测试也测试性能、可靠性，并检查该系统并没有做不应该做的事，例如产生不期望的输出等。Andrea（2007）建议如何扩展测试工具将系统测试的某些

方面和 TDD 集成在一起。

测试驱动开发已经证明对于小的或中等规模的项目来说是一个成功的方法。一般来说，采用这个方法的程序员都很乐意用它并发现它是一种更有生产率的软件开发方法（Jeffries 和 Melnik，2007）。在一些试验中，它已证明会提高代码质量；在其他试验中，结果尚无定论。然而，没有证据证明 TDD 会降低代码质量。

8.3 发布测试

发布测试是为开发组以外的用户使用的系统的一个特殊版本所做的测试过程。通常，系统发布版本是为了客户和使用者。但在一个复杂项目中，发布版本可能是为了正在开发有关系的系统的另一个小组。对于软件产品来说，发布版本可能为了将来出售它的产品管理者而准备的。

在开发过程中，发布测试和系统测试之间有两个重要的区别：

1. 一个独立的与系统开发无关的小组应该负责发布测试。
2. 开发组的系统测试的重点是在系统中发现错误（缺陷测试）。发布测试的目标是检查系统符合它的需求描述，并且是可以对外销售（有效性验证测试）。

这个过程的主要的目标是增加供应商对系统能良好地使用的信心。如果系统满足需求的话，它就可以作为一个产品或移交给客户的软件发布出去。因此，发布测试必须证明系统具有指定的功能、性能和可依赖性，在常规操作下不会出错。它应该将所有的系统需求考虑在内，而不仅仅是系统最终用户的需求。

发布测试通常是一个黑盒测试过程，测试从系统描述导出。系统被作为一个黑盒子，它的行为只能通过输入和与它相应的输出来确定。这种测试的另一个名称叫做“功能测试”，因为测试者只关心系统的功能并不关心软件的实现。

8.3.1 基于需求的测试

好的需求工程实践的一个总的原则是需求应该是可测试的。也就是说，需求应该写得容易从中导出测试设计。测试者于是可以检查需求是否满足了。因而，基于需求的测试是一种系统化的测试用例设计方法，我们考虑每一个需求并得到它的一组测试。基于需求的测试是有效性验证测试而不是缺陷测试——我们要努力证明系统已经正确地实现了它的需求。

例如，考虑 MHC - PMS 的需求（在第 1 章中介绍的），这些都与药物过敏审查有关。

如果知道一个病人对任何特殊药物过敏，那么有该药物的处方将导致向系统用户发出警告消息。

如果开处方的医生选择忽略过敏警告，那么他/她应当提供一个忽略的原因。

为了检查是否满足这些需求，我们可能需要开发一些相关的测试：

1. 建立一个没有已知过敏史的病历记录。开一个会出现某已知过敏症的处方。检查系统没有发出警告信息。
2. 建立一个有已知过敏史的病历记录。开一个会出现病人对此过敏的处方，并检查警告信息由系统发出。
3. 建立一个病历记录，其中包括两个或两个以上过敏药物。开两个处方分别带有其中一种过敏药物，并检查系统为每种药物发出了正确的警告信息。
4. 为病人开带有两个过敏药物的处方。检查发出的两个正确的警告信息。
5. 开一个带某种过敏药物的处方，这种药物发出警告，医生拒绝了这个警告。检查系统允许用户提供信息解释警告被拒的原因。

从中我们可以看到，测试一个需求不意味着只写一个测试。我们通常要写很多个测试来保

证对此需求的覆盖。我们也应该为基于需求的测试保持我们的追踪记录，它将测试和正在测试的特殊需求联系起来。

8.3.2 情景测试

情景测试也叫脚本测试，或场景测试是发布测试的一个方法，设计典型的使用场景并使用这些来为系统开发测试用例。一个情景是一个故事，这个故事描述了系统使用的一种方式。情景测试应该现实，并且真实的系统用户应该能将它和自己的工作联系起来。如果你使用过情景作为需求工程过程的一部分（见第4章），那么你就能够复用这些作为测试情景。

在关于情景测试的一篇论文中，Kaner（2003）建议：一个情景测试应该是一个叙述性的故事，这个故事是可信的和适度复杂的。它应该激发信息持有者；也就是说，他们应该和情景有共鸣，并且相信系统通过此测试是很重要的。他还建议它应该易于评估。如果系统有问题，则发布测试组应该能够看出来。作为 MHC-PMS 的一个可能的情景的例子，图 8-10 描述了该系统在出诊中可能的一个使用方式。

1. 通过登录到系统进行身份验证。
2. 下载和上传特定病人的病历到笔记本电脑。
3. 家庭出诊时间安排。
4. 在移动设备上加密和解密病历。
5. 记录检索和修改。
6. 连接包含副作用信息的药物数据库。
7. 电话提示系统。

如果我们是一个发布测试者，我们通过这个情景，看看 Kate 扮演的角色，并且观察对不同的输入系统是如何回应的。作为“Kate”，可能会故意地犯错误，例如输入错误的密钥短语来解密记录。这是检查系统对错误的响应。我们应该仔细地注意引起的问题，包括性能问题。如果一个系统太慢，这将会改变使用它的方式。例如，如果加密一个记录时间过长，那么着急的用户可能会跳过这个阶段。如果他们丢失笔记本电脑，那么未经授权的人可以查看病人的病历。

当我们使用基于情景的方法时，我们通常在同一场景测试几个需求。因此，除了检查单个需求外，我们也应该检查需求的组合不会引起问题。

Kate 是一个专门从事精神健康护理的护士。她的责任之一是访问在家的病人来检查他们的治疗是否有效，以及确认他们没有遭受到药物的负面作用的危害。

家访的一天，Kate 登入到 MHC-PMS，并使用它来打印她那天的家访时间表，还有关于她要家访的病人的所有信息。她请求下载这些病人的记录到她的笔记本上。提示她用密钥短语来加密笔记本电脑的记录。

她家访的一个病人叫 Jim，他是在进行抑郁症的治疗。Jim 感觉药物可以帮助他，但是他认为这药物有让他晚上睡不着觉的副作用。Kate 察看他的记录，系统提示她输入密钥短语来解密记录。她检查了开的药物，并查询它的副作用。失眠是一个已知的药物副作用，因此她在 Jim 的记录里标注了这个问题，并且建议 Jim 去诊所改变药物。他同意了，因此 Kate 记录了一个电话提醒以便记住当她回到诊所后帮助他预约一个医生然后打电话给 Jim。她结束了谈话，并在系统重新加密 Jim 的记录。

完成她的谈话后，Kate 返回诊所，并上传家访病人的记录到数据库中。系统生成 Kate 的一个联络人的电话清单，这些是需要后续追踪调查和提供诊所预约的病人。

图 8-10 MHC-PMS 情景的使用

8.3.3 性能测试

一旦一个系统已经完全集成，就可以测试它的总体特性了，这些总体特性的例子有性能和

可靠性。性能测试必须设计以保证系统可以处理预期的负荷。这通常包括对一系列的测试的规划，这里的测试是要让负荷稳定地增长直到系统性能不可接受为止。

和其他测试类型相比，性能测试既关心对系统满足其需求的证明，也要关心发现系统的问题和缺陷。为了测试系统是否达到了它的性能需求，我们必须建立一个操作概要。操作概要（见第15章）是一组反映由系统所处理的实际混合工作的测试。因此，如果系统90%的交易是A型的，5%的交易是B型，剩余的有C、D、E型，那么我们就必须设计一个操作简档，这样绝大多数测试是A类型的。否则，我们将不会得到系统操作性能的精确测试。

当然这个方法不一定是缺陷测试的最好的方法。经验告诉我们发现缺陷的最有效的方法是设计系统边界的测试。在性能测试中，这意味着给系统以压力，指的是通过制造软件设计边界外的需要来测试系统。这个也被称为“压力测试”。例如，我们正在测试一个交易处理系统，这一系统被设计成处理每秒钟300个交易，然后我们逐步地以每秒超过300个交易业务来增加系统负荷，直到在系统的最大设计负荷之上是完好的或系统失败。这种类型的测试有两个功能：

1. 它能测试系统的失败行为。当测试负荷达到系统预期的最大负荷时，系统会产生一些特别的反应。在这种情况下，最重要的是系统失败不应该引起数据崩溃或者是意想不到的用户损失。压力测试审查过载系统引起的故障保护而不是在此负荷下崩溃。
2. 它给系统以压力，可以让系统暴露那些在正常情况下不会暴露的缺陷。虽然有人会认为这种缺陷在正常情况下根本不会导致系统失败，但这种压力测试可能模拟一些正常情况下的不寻常组合。

压力测试尤其与基于处理器网络的分布式系统有关。在大负载情况下，这些系统通常性能急剧下降。不同的处理器间需要交换数据，由网络来协调这些数据，某些处理器需要等待来自其他处理器的数据，因而整个网络变得越来越慢。当性能下降时，压力测试可以帮我们发现问题，这样我们就可以对系统增加检查来拒绝超出此临界点的交易。

8.4 用户测试

用户或客户测试是测试过程中的一个阶段，在这个阶段，用户或客户提供输入和系统测试建议。这可能会是一个正式的对外部供应商提供的系统的测试，或是一个非正式的过程，用户测试一个新的软件产品来看看他们是否喜欢或这个产品是否能符合他们的需求。用户测试是必不可少的，即使全面的系统测试和发布测试已完成。原因是来自用户工作环境的因素对一个系统的可靠性、性能、可用性以及健壮性有很大的影响。

对于一个系统开发者来说，他实际上是不可能复制该系统的工作环境的，因为在开发环境中的测试难免是模仿的。例如，假设一个用于医院的系统用在一个诊所的环境中，其他事情照常进行，如病人急救，与亲人交谈等。这些都会影响一个系统的使用，但是开发者不可能在测试中包括这些内容。

实际上，存在3种不同的用户测试类型：

1. α 测试，软件的用户与开发小组一起在开发者的地点测试这个软件。
2. β 测试，该软件的版本是提供给用户让他们进行试验，并向开发者提出他们所发现的问题。
3. 接收测试，客户测试系统来决定他们是否愿意从系统开发者那里接收系统并在客户环境中部署。

在 α 测试中，系统一边开发，用户和开发者一边一起测试这个系统。这意味着用户可以发现那些对开发测试小组不是很明显的问题和情况。开发者真的只能按照需求来开发，但是这些需求通常不能反映其他影响软件使用的因素。用户因此可以提供实践的信息，这样可以帮助设计更多符合实际的测试。

α 测试经常在开发的软件产品作为套装系统来出售的时候使用。这些产品的用户很乐意参与到 α 测试过程中，因为这可以让他们早点了解有关新系统功能的信息。 α 测试也降低了软件非预期的变更会对它们的市场带来破坏性影响的风险。然而， α 测试在客户软件正在开发的时候也会用到。敏捷的方法，例如极限编程主张在开发过程中用户的参与，使用者应该在设计系统测试中起到关键作用。

β 测试发生的早期，有时该系统未完成，软件系统的版本是提供给客户和用户进行评估的。 β 测试可能是从早期使用该系统的客户中选出来组成一个小组。或者，该软件可以公开地供对这个有兴趣的人使用。 β 测试最常用在那些在许多不同的环境中使用的软件产品（相对的是定制系统，一般都是在确定的环境中使用的）。软件开发者不可能了解和复制软件使用的所有环境。因此 β 测试对于发现软件和使用该软件的环境特性之间的交互问题是必不可少的。 β 测试也形成一个市场——客户可以学习他们的系统，并且了解系统可以为他们做什么。

接收测试是定制系统开发的一个固有的部分。发布测试之后就是接收测试。它需要客户正式测试一个系统来决定是否可以从开发商那里接收这个系统。接收意味着要为该系统付费。

在接收测试过程中有 6 个阶段，如图 8-11 所示。它们是：

1. **定义接收准则** 这个阶段理想情况下应该是发生在签订系统合同之前的过程早期。接收准则应该是系统合同一部分，并得到客户和开发者的同意。然而实际上，在早期阶段定义接收准则是很困难的。详细的需求描述可能是不可得的，在开发过程中需求可能会有重大变化。

2. **计划接收测试** 这涉及确定接收测试所需要的资源、时间和预算，以及做一个测试计划表。这个接收测试计划也应该讨论对需求所需的覆盖和系统功能测试的顺序。它应该确定测试过程的风险，例如系统崩溃、性能不足，并且讨论这些风险如何得到缓解。

3. **导出接收测试** 一旦接收准则确定后，就需要设计测试来检查系统是否可以接收。接收测试目的应该是测试系统的功能性和非功能性特性（例如性能）。理想情况下，接收测试应该提供完整的系统要求覆盖。实际上，很难建立完全客观的接收标准。经常会有关于一个测试是否证明了它确实符合了某个准则的争论。

4. **运行接收测试** 经过同意的测试在系统上执行。理想情况下，这个应该发生在系统使用的真实环境中，但是这可能是破坏性和不实际的。因此，必须建立一个用户测试环境来运行这些测试。使这个过程自动运行是很困难的，因为接收测试的一部分可能包含测试最后用户和系统之间的交互。可能需要对最终用户有一些培训。

5. **协商测试结果** 所有定义的接收测试全部通过并且系统没有任何问题，这是不太可能的。如果是这种情况的话，则接收测试就完成了，并且系统可以交付了。更常见的是，会发现一些问题。在这种情况下，开发者和客户必须协商来决定该系统是否能够很好地投入使用。他们还必须商定开发者对已确定问题的回应。

6. **拒绝/接收系统** 这个阶段包含开发者和客户之间的一个会议，这个会议来决定这个系统是否可以接收。如果该系统不足以来使用，则需要进一步的开发来解决这个已确定的问题。一旦完成，接收测试阶段就循环进行。

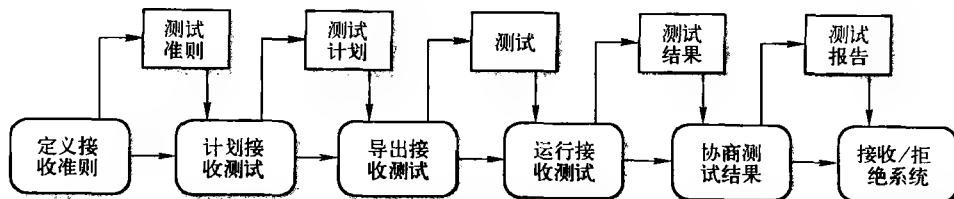


图 8-11 接收测试过程

在敏捷方法中,例如极限编程,接收测试有不同的意思。原则上,它认可这样一个概念,即用户应决定是否接受一个系统。然而,在极限编程中,用户是开发组的一部分(例如,他是一个 α 测试人员),并且以用户故事的形式提供系统需求。他或她也负责定义测试,这个测试决定开发的软件是否支持用户的故事(情景)。测试是自动执行的,直到故事接收测试通过,否则开发不会继续进行。因此没有独立的接收测试活动。

正如第3章中所讨论的,用户参与的一个问题是要确保嵌入在开发小组的用户是一个“典型”的用户,他知道系统使用的一般知识。找到一个这样的用户是很困难的,因此接收测试可能不是实际情况的真实反映。还有,自动化测试的需求严重限制了交互式测试系统的灵活性。对于这样的系统,接收测试可能需要一组最终用户来使用这个系统,好像这是他们每天工作的一部分一样。

我们可能认为接收测试是一种明确的合同问题。如果一个系统没有通过它的接收测试,则这个系统不应该接收,并且不应付费。然而,事实是很复杂的。客户想尽可能快地使用这个软件,早一天部署就可以早一天盈利。他们也许已购买新的硬件,已经为此进行了人员培训,也为此改变了他们的流程。他们也许乐意接收这个软件,不管它有什么样的问题,因为不使用这个软件的代价要比带有问题来工作的代价更大。因此,协商的结果可能是有条件地接收这个系统。客户也许接收这个系统,就可以开始部署了。系统的提供商同意修复紧急问题,以及尽快给客户一个新的版本。

要点

- 测试只能证明系统中存在错误。它不能说明系统中不再有缺陷。
- 开发测试是软件开发小组的责任。独立的测试小组应该在系统发布给客户之前负起测试系统的责任。在用户测试过程中,客户或系统使用者提供测试数据,以及确定测试成功。
- 开发测试包括单元测试,即测试单个对象和方法;组件测试,即测试相关的一组对象;系统测试,即测试部分或完整的系统。
- 测试软件时,我们应该根据经验和原则把软件打散来选择测试案例的类型,选择那些在其他系统中已经证明是能有效地发现缺陷的那些测试案例类型。
- 有可能的话,我们应该编写自动化测试。将这个测试嵌入到程序中,这样的程序可以每次当系统一有变化就运行。
- 测试优先开发是开发的一种方法,在代码测试之前就已经写好测试。做小的代码更改和进行代码重构,直到所有测试成功执行。
- 情景(脚本)测试是非常有用的,因为它可以复制系统的使用。它包括设计一种典型的使用场景,并以此来导出测试案例。
- 接收测试是一个用户的测试过程,目的是决定这个软件是否足以进行部署。以及能否用在它的实际操作环境中。

进一步阅读材料

《How to design practical test cases》 这是一篇有关测试案例设计的入门指导文章,作者来自一家日本公司,该公司在生产优质软件方面极负盛名(T. Yamaura, IEEE Software, 15 (6), November 1998)。http: //dx. doi. org/10. 1109/52. 730835。

《How to Break Software: A Practical Guide to Testing》 这是一本软件测试的实用书,而不是一本理论教材。作者给出了很多基于经验的有关设计测试的准则,这些准则对于发现系统缺陷是很有效的(J. A. Whittaker, Addison-Wesley, 2002)。

《Software Testing and Verification》 这是IBM系统期刊的一个特别专辑,包括一系列有关测

试的论文，其中包括一个很好的综述，以及有关测试量度和测试自动化的几篇论文（IBM Systems Journal, 41 (1), January 2002）。

《Test-driven development》 这个关于测试驱动开发的专辑包括一个关于 TDD 的很好的回顾和关于 TDD 如何在不同类型的软件中使用的经验文章（IEEE SOFTWARE, 24 (3) May/June 2007）。

练习

- 8.1 解释在系统交付给客户之前，为什么对于程序来说没有一点缺陷是没有必要的。
- 8.2 解释为什么测试只能检测错误的存在，而不是证明它的不存在。
- 8.3 一些人认为开发者不应该参与测试他们自己的代码，所有的测试应该是独立小组的责任。给出支持和反对开发者进行测试的观点。
- 8.4 假设请求我们测试 Paragraph 对象中一个叫做“catWhiteSpace”的方法，在段落中，使用单个的空白字符代替连续的空白字符。找出这个例子的测试划分并导出对 catWhiteSpace 方法的一个测试集合。
- 8.5 什么是回归测试？解释怎样使用自动测试和测试框架（如 JUnit）简化回退测试。
- 8.6 MHC-PMS 是通过调整一种现成的信息系统来构建的。你怎样认为测试一个这样的系统和测试一个用面向对象的语言（例如 Java）开发的软件之间的区别。
- 8.7 写一个情景用来帮助设计气象站系统的测试。
- 8.8 我们如何理解术语“压力测试”？如何对 MHC - PMS 进行压力测试。
- 8.9 在测试过程的早期阶段，用户参与到发布测试的优势是什么？用户参与有劣势吗？
- 8.10 系统测试的一个普遍的测试方法是测试系统直到测试预算耗尽，然后将系统交付给客户。对于要交付给外部客户的系统来说，讨论这个方法的道德问题。

参考书目

- Andrea, J. (2007). 'Envisioning the Next Generation of Functional Testing Tools'. *IEEE Software*, 24 (3), 58-65.
- Beck, K. (2002). *Test Driven Development: By Example*. Boston: Addison-Wesley.
- Bezier, B. (1990). *Software Testing Techniques, 2nd edition*. New York: Van Nostrand Rheinhold.
- Boehm, B. W. (1979). 'Software engineering; R & D Trends and defense needs.' In *Research Directions in Software Technology*. Wegner, P. (ed.). Cambridge, Mass.: MIT Press. 1-9.
- Cusamano, M. and Selby, R. W. (1998). *Microsoft Secrets*. New York: Simon and Shuster.
- Dijkstra, E. W., Dahl, O. J. and Hoare, C. A. R. (1972). *Structured Programming*. London: Academic Press.
- Fagan, M. E. (1986). 'Advances in Software Inspections'. *IEEE Trans. on Software Eng.*, SE-12 (7), 744-51.
- Jeffries, R. and Melnik, G. (2007). 'TDD: The Art of Fearless Programming'. *IEEE Software*, 24, 24-30.
- Kaner, C. (2003). 'The power of 'What If . . .' and nine ways to fuel your imagination: Cem Kaner on scenario testing'. *Software Testing and Quality Engineering*, 5 (5), 16-22.
- Lutz, R. R. (1993). 'Analyzing Software Requirements Errors in Safety-Critical Embedded Systems'. RE'93, San Diego, Calif.: IEEE.
- Martin, R. C. (2007). 'Professionalism and Test-Driven Development'. *IEEE Software*, 24 (3), 32-6.
- Massol, V. and Husted, T. (2003). *JUnit in Action*. Greenwich, Conn.: Manning Publications Co.
- Prowell, S. J., Trammell, C. J., Linger, R. C. and Poore, J. H. (1999). *Cleanroom Software Engineering: Technology and Process*. Reading, Mass.: Addison-Wesley.
- Whittaker, J. W. (2002). *How to Break Software: A Practical Guide to Testing*. Boston: Addison-Wesley.

软件进化

目标

本章旨在解释为什么软件进化是软件工程中一个重要组成部分，讲解软件进化过程。读完本章，你将了解以下内容：

- 了解变更是不可避免的。如果要使软件系统继续有用，那么软件开发和软件进化将集成在螺旋模型中。
- 了解软件进化过程以及作用于这些过程上的因素。
- 学习不同类型的软件维护和影响维护费用的因素。
- 了解如何对遗留系统进行评估，从而决定是应该抛弃、维护、再工程还是更换这些系统。

软件的开发不会因为系统的交付而停止，它贯穿系统的整个生命周期。在系统开发完成后，如果要使其继续有用，那么对它进行修改是不可避免的。由于业务变更和用户期待的改变，使得对已有系统的新需求浮现出来。由于种种原因，软件的某些部分需要修改。如修改正运行中发现的错误，适应新的运行平台，提升性能或其他的非功能特性。所有这些都意味着在系统交付后，软件系统总是在不断进化以满足变更的需求。

由于机构完全依赖它们的软件系统，并且上面可能已投入了数百万美元，因此软件进化对它们来讲是非常重要的。系统已经成为机构的重要经营资产，必须投资进行系统变更以保持其价值。通常，大多数大型公司在维护系统上的开支要比在系统开发上的开支多很多。基于一项非正式的工业调查结果，Erlikh (Erlikh, 2000) 认为 85% ~ 90% 的软件花费是进化花费。其他的调查也显示大约 2/3 的软件花费是进化花费。无疑，对于所有 IT 公司来说软件变更的开支都占据了预算的很大一部分。

变更的用户需求，软件的错误报告，或是一个软件系统环境中其他系统的更改都有可能成为软件进化的原因。Hopkins 和 Jenkins (2008) 创造出一个术语“布朗菲尔德软件开发 (brown-field software development)”用以描述软件系统的情况，若软件系统处于依赖其他许多软件系统的环境中，则需要其做出改进和控制。

因此，一个系统的进化不能被认为是孤立的。对环境的改变会导致系统的改变，从而可能进一步导致对环境的改变。当然，在一个“系统丰富”的环境中的系统进化通常会增加进化的困难和成本。认识和分析了系统本身提出变化的影响，你还可能不得不去评估这会对运行环境中的其他系统可能产生怎样的影响。

令人满意的软件系统通常有一个很长的生命周期。例如，大型的军事或者基础架构系统，像航空交通管制系统，可能拥有 30 年或者更长时间的生命周期。商业系统的生命周期通常也会在 10 年以上。软件的成本很高，所以一个公司会使用一个软件系统很多年来收回前期对软件产品的投资。显然，已安装的系统，随着业务和它的环境的改变，其需求也随之改变。因此，系统的新版本以及加入的改变和更新，通常会定期发布新版本。

因此，软件工程是一个贯穿系统生命周期的由需求、设计、实现、测试组成的螺旋过程 (见图 9-1)。你开始于系统的第 1 个版本的创建。一旦交付使用，变更提出，则版本 2 的开发立刻开始。事实上，甚至于在系统部署之前，进化的需求可能已经变得很明显，以至于在目前的版

本发布之前，软件的后继版本已就在开发中了。

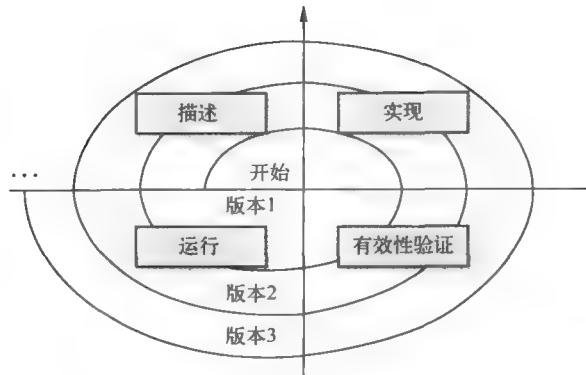


图 9-1 开发和进化的螺旋模型

这个软件进化模型是针对一个专门机构既负责最初的软件开发又负责以后的软件进化这种情况的。大部分打包的软件产品是按这种方式开发的。对于定制软件，通常使用另一种不同的方法。一个软件公司为客户开发软件，然后由客户自己的开发团队接管这个系统，即由他们负责软件进化。还有另一种选择是，软件用户和另外一家不同的公司签订一个单独的合同，要求其负责系统的支持和进化。

在这种情况下，螺旋进程经常是不连续的。需求和设计文档不能从一个公司传递到另一个公司。公司可能合并或重组，继承来自其他公司的软件，于是发现需求和设计文档都必须进行变更。当从开发到进化的转换不是无缝衔接时，软件移交之后的变更过程被称为软件维护。如在本章的后面将会谈到的，维护包括额外的过程活动，例如，除正常的软件开发活动外的程序理解。

Rajlich 和 Bennett (2000) 提出一个软件进化生命周期的另一个替代视图，如图 9-2 所示。在这个模型中，他们把进化和服务区别开来。进化是指涉及软件体系结构和功能性重大改变的阶段，而在服务阶段，只是对软件做一些相对小的和必不可少的一些改变。

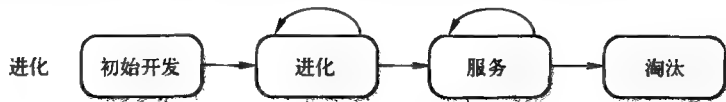


图 9-2 进化与服务

在进化过程中，软件在顺利地使用并且有一个连续的关于需求变化的提议。但是，随着软件被更改，它的结构也在退化，更改的成本也变得越来越来。这种情况通常发生在使用数年以后，其间也有其他环境的改变，比如说硬件或者操作系统。在生命周期的某些阶段，软件到达了一个转变点，软件需要做出重大的变化，实现新的需求，并且变得越来越不符合成本效益。

在这个阶段，软件从进化转向服务。在服务阶段，软件仍然是可用的并且一直在使用着，只有一些小的局部的调整需要做。在这个阶段，公司经常会考虑怎样将软件更换掉。在最后的阶段，逐步淘汰，软件可能还在使用，但是不会有进一步的改动需要被实现。用户需要去想办法绕过所发现的任何问题。

9.1 进化过程

软件进化过程在相当程度上依赖于所维护的软件的不同类型和参与开发过程的机构和人。在一些机构中，进化可能是一种非正式的过程，变更请求大部分来自于系统用户和开发者的交流。在另外一些公司，这却是一个在每个阶段都产生结构化文档的正式过程。

在所有的机构中，系统变更建议都是系统进化的动力。这些变更建议可能包括在发布的版本中还没有实现的已有需求、新的需求请求、系统所有者的补丁要求和来自系统开发团队的改进软件的新想法和建议。变更识别的过程和系统进化是循环和持续的，并且贯穿于系统的生命周期（见图9-3）。

变更提议应当与需要实现这些提议的系统的组成部分联系起来。这样就允许我们估计成本和影响。这是通常的变更管理过程的一部分，它还应当保证在每一个系统版本中各组成部分有其正确的版本。我们将在第25章中详细讲述变更和配置管理。

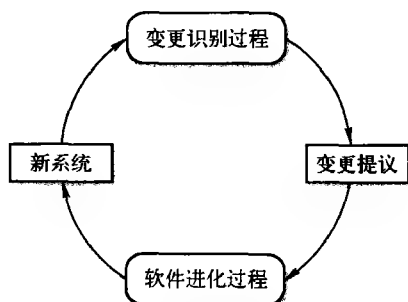


图9-3 变更发现和进化过程

图9-4 摘自 Arthur (1988)，展示了进化过程的概况。

进化过程包括变更分析的基础活动、版本规划、系统实现和对客户发布。通过对这些变更的花费与影响的评估，来发现系统在多大程度上受到影响以及实现变更可能花费多少。如果变更建议被接受，那么系统的新版本就在规划中了。在版本规划中，所有的变更建议（缺陷修补、改写和新功能）都将得到考虑。随后做出决定在系统的下一版本中实现哪些变更。对这些变更加以实现和验证，发布新版本。在下一个版本中重复这个过程，形成一个新的变更建议集合。

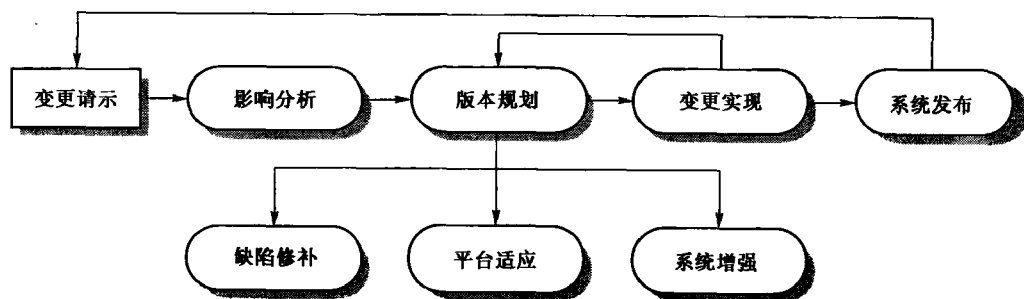


图9-4 软件进化过程

你可以把变更实现的过程看成是一个开发过程的迭代过程，在此迭代过程中完成对系统修改版本的设计、实现和测试。只是，这里的一个重要的区别是变更实现的最初阶段是对程序的理解，特别是当源系统的开发者不承担变更实现的责任时。在这个阶段，你要了解程序是怎样构造和怎样实现它的功能的。在实现一个变更时，我们要利用以上的理解来确保实现的变更不会对现有系统产生不利影响。

理论上，在此过程的变更实现阶段应该修改系统的描述、设计和实现来反映对系统的变更（见图9-5）。提出反映系统变更建议的新需求，并对此加以分析和验证。对系统组件重新设计、实现和测试。适当的话，还要构造变更建议的一个原型作为变更分析的一部分。

在进化过程中，要详细分析需求，因此往往在变更分析的早期阶段原本不明显的一些变更要求逐渐浮现出来。这意味着所提出来的变更有可能要修改，并且在实现前需要进一步的与客户讨论。

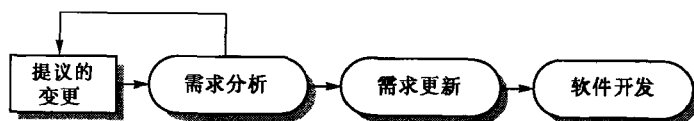


图9-5 变更实现

变更请求有时关系到需要被立刻解决的一些系统问题，有下面几个原因使得这些紧迫的变更会发生：

1. 如果有严重的系统缺陷发生，那么必须修补而使正常的操作继续。
2. 如果系统操作环境的变更中有不期望的情况发生，那么就会破坏正常的操作。
3. 如果系统上运行的业务有未预料到的改变发生，比如有新的竞争对手出现或者有新的法律生效并影响到了系统。

在这些情况下，就需要做出快速的变更，也意味着你不能遵循正常的变更分析过程。不是按正常的过程去修改需求和设计，而是要对程序做紧急的修补来解决突发问题（见图 9-6）。这样做的危险是使得需求、软件设计和代码逐渐变得不一致性。尽管你打算记录需求和设计方面的变更，却可能又有一个急需的修补在等你完成。它的优先权高于文档记录。最后，最初的变更被遗忘了，系统文档与代码再也不能一致了。

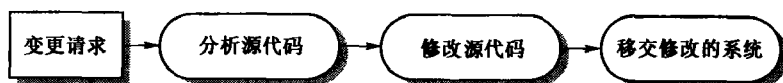


图 9-6 紧急修补过程

紧急系统修补通常需要尽可能快地完成。你应当选择一种快速的可行方案，而不是保证系统结构最好的方案。这就加速了软件的老化过程，并使未来的变更计划更困难，维护费用上升。

理想情况是，在紧急代码修补完成之后，虽然代码缺陷已经修复，但应该继续保持变更请求突出显示。这样就可以在经过深入的分析后，让它得到重新实现，这一次的实现将是更加仔细。当然，修补代码可能再次得到利用。当我们有更多的分析时间的话，就有可能找到此问题的另一个更好解决方案。但是实际上，这些变更的优先级不可避免地是较低的。它们通常被遗忘，并且当执行完其他系统变更之后，再重新做紧急修补几乎是不太可能的。

敏捷方法和过程（在第 3 章中讨论过）它们也许会在程序进化和程序开发中用到。实际上，因为这些方法是基于增量开发的，将敏捷开发向交付后进化的转变应当是无缝衔接的。像自动回归测试这样的技术在系统更改时是有用的。变更可能表达为用户的故事脚本，客户参与能够把运行系统中需要的变更优先执行。总之，进化就是简单地继续敏捷开发的过程。

然而，当一个开发团队向另外一个负责进化的团队交接时，有两种问题则可能出现：

1. 开发团队运用了敏捷方法，但是进化团队却不熟悉敏捷方法而选择了一个计划驱动的方法。进化团队可能期望详细的文档来支持进化工作的进行，而这恰恰是敏捷方法所没有提供的。可能没有关于系统的一个明确的描述以供变更时使用。

2. 当计划驱动的方法被用于开发时，而进化团队选择使用敏捷方法。这种情况下，进化团队可能不得不从头开发自动化的测试，而且不会有像敏捷开发过程中所期待的系统代码重构和简化。这样，在采用敏捷开发过程之前，可能要求使用一些再工程方法来提升其代码质量。

Poole 和 Huisman（2001）的报告提供了在使用极限编程来维护最初是使用计划驱动方法来开发的大型系统的经验。在对系统的再工程以提升其结构性之后，XP 在维护过程中使用得非常成功。

9.2 程序进化的动态特性

程序进化的动态特性就是对系统变更的研究。开始于 20 世纪 70 年代和 80 年代，Lehman 和 Belady（1985）为了理解更多的软件进化特性，进行了一些关于系统变更方面的实验研究。这项工作一直进行到 20 世纪 90 年代初，Lehman 和其他人开始研究进化过程中反馈的重要性

(Lehman, 1996; Lehman 等, 1998; Lehman 等, 2001)。经过这些研究, 他们提出了关于系统变更的 Lehman 定律 (见图 9-7)。

定 律	描 述
持续变更	现实环境中的程序必须进行变更, 否则将失去在相应环境中的作用
不断增长的复杂性	随着程序变更的发生, 其结构逐渐变得更加复杂, 需要有额外的资源来保持和简化结构
大型程序进化	程序进化是一个自我控制过程。系统属性, 如规模、两个版本间的时间间隔以及报告的错误数量等, 对每个系统版本基本上是不变的
机构稳定性	在一个程序的生命周期中, 其开发速度几乎是一个常数, 并且与所投放到系统开发中的资源无关
保持亲密性	在一个系统的生命周期中, 每个版本的变更增量都接近常数
持续的增长	系统提供的功能需要连续地增加, 以保证用户满意
降低质量	系统的质量将出现下滑, 除非在其运行环境中能不断适应变化
反馈系统	进化过程结合多智能体和多环反馈系统, 我们必须把它看成一个反馈系统来取得效果显著的产品改善

图 9-7 Lehman 定律

Lehman 和 Belady 声称这些定律很可能对所有类型的大型软件系统 (他们称为 E 类系统) 都适用。需求在这些系统中的变化反映出业务需求的变更。系统的新版本对于系统提供的商业价值来说是必不可少的。

第 1 条定律的内容是: 系统维护是一个不可避免的过程。当系统的环境发生改变, 新的需求就会浮现, 系统就必须修改。当修改后系统重新投入使用, 又会促进环境的改变, 于是进化过程又开始了。

第 2 条定律的内容是: 随着系统的改变, 其结构在退化。避免退化发生的唯一方法是在预防性维护上的投入, 在维护软件结构上下工夫, 而不是向系统中增加新的功能。很显然, 这样做就等于在实现必要的系统变更成本上又要增加额外的开支。

第 3 条定律可能是 Lehman 定律中最有趣同时也最有争议的。它认为大型系统自身的动态特性是在开发过程的早期阶段建立的。这决定了系统维护过程的总的趋势以及系统变更可能次数的极限。Lehman 和 Belady 认为, 该定律是结构化因素和机构因素作用的结果, 结构化因素影响和制约系统的变更, 机构因素影响进化过程。

结构化因素影响的第 3 条定律来自于大型系统的复杂性。当你更改和扩展程序时, 系统的结构性就会下降。任何类型的系统 (不仅仅是软件) 都是这样, 当你为了达到某种目的改写一个结构时, 系统的结构性就会下降。这种下降, 如果没有被检查到, 会使得以后对程序的变动越来越困难。做出小的改动会降低结构性, 同时也会减少其导致严重系统依赖性问题的风险。如果你试图做出大的改动, 就很可能引进新的错误。这些错误会阻碍进一步的程序变更。

机构因素影响第 3 条定律反映出大型系统通常是由大型机构设计出来的。这些机构的内部管理人员负责设定每个系统的变更预算并控制决策过程。机构需要对变更的价值和风险以及其所需要的成本做出决策。这些决策通常很花时间, 有时甚至变更的决定比变更的实现花费更长的时间。因而系统的变更频度是受机构决策过程的速度制约的。

Lehman 的第 4 条定律的内容是: 绝大多数大型程序设计项目是在一种称之为“饱和”状态下运作的。这就是说, 资源和人员上的变化对系统长期演化的影响是不易察觉的。当然, 在第 3 条定律中也提到了这一点, 在那里讲到程序演化很大程度上是独立于管理决策的。这条定律则

认为大型软件开发团队通常是低效的,沟通成为团队工作的重要制约因素。

Lehman 的第 5 条定律是关于在每个系统版本中的变更增量。向系统中添加新功能不可避免会把新的缺陷引入到系统中。在每个版本中增加的功能愈多,缺陷将会愈多。因此,在一个系统版本中一个大的功能增量将意味着后面不得不紧跟着一个版本来修补系统新引入缺陷,相对来说在该版本中新的功能反而较少。这个定律告诉我们:在为每个版本中大的功能增量做预算时应该考虑还需要修补缺陷。

前面的 5 条定律是 Lehman 最初的建议,后面的定律是在以后的工作中增加的。第 6 条和第 7 条定律很相似,基本的意思是说:软件的用户将变得愈加的不满意,除非软件得到维护并且有新功能加入进来。最后的定律反映了在反馈过程上的新近工作,尽管现在仍然不清楚怎样把它应用到实际的软件开发中去。

Lehman 的结论总的来讲是符合实际的,在规划维护过程中应该作为参考。但在有些情况下,可能会出于经营上的考虑而被忽略掉。举例来说,出于市场的原因,可能需要在在一个版本中加进多个较大的系统变更,结果就需要后续的一个或多个版本专门用来更正错误。我们常常能在个人电脑软件中看到这一点,紧紧跟随着一个应用的一个较大的新版本的是一个补丁更新。

9.3 软件维护

当软件交付后,软件维护就成为软件变更的一个常规过程。变更可以是一种更正代码错误的简单变更,可以是更正设计错误的较大范围的变更,还可以是对描述错误进行修正或提供新需求这样的重大改进。变更的实现是修改已有的系统组件以及在必要的地方添加新组件到系统中。

有 3 种不同类型的软件维护:

1. **修补软件缺陷** 通常改正代码错误费用相对较低,改正设计错误费用就高得多,因为要重写很多程序组件。需求错误的更正费用更高,因为对系统进行大量的重设计是必须的。

2. **使软件适应不同操作环境** 在系统环境的某些方面发生改变的时候,需要进行这种类型的维护。环境上的改变包括硬件变化、操作系统平台的变化或其他的支持软件发生变化。为了适应这些环境变化必须修改应用系统。

3. **增加或修改系统功能** 当系统需求随着机构因素或业务改变而变更的时候,这种类型的维护就是必要的了。这时系统需要变更的范围通常要比其他类型的维护要大得多。

在实际过程中,这些不同类型的维护之间没有一个明确的界限。在使软件适应一个新环境的时候,可能需要增加新的功能来充分利用环境提供的服务。软件缺陷可能是因为系统在一种未预料的方式下使用才得以暴露,修正这类缺陷的最好方法是改变系统以适应它们的工作方式。

尽管维护的不同类型得到了普遍认可,但有时可能会使用其他不同的分类名称。人们广泛使用的“纠正性维护”这个说法是指缺陷修补维护;“适应性维护”有时是指为适应新环境所做的维护,而有时又是指对新需求的适应性维护;“完善性维护”有时指实现一个新需求来完善软件,而有时又指保留系统的功能但改善结构和性能。由于这些名称涵义的不确定性,本章尽量避免使用这类术语。

有一些软件维护方面的研究,它们着眼于维护和开发之间的关系,以及不同维护活动之间的关系(Krogstie 等, 2005; Lientz 和 Swanson, 1980; Nosek 和 Palvia, 1990; Sousa, 1998)。因为术语用法方面的不同,这些研究的具体细节无法比较。尽管改变出现在技术上和不同的应用领域,可是自从 20 世纪 80 年代以来在进化方面投入的百分比很少有改变。

调查大多表明,软件维护成本在信息科技预算中比新的开发占据了更高的比例(大约维护占到了 2/3,开发占 1/3)。调查同样表明,更多的维护预算是用在了实现新需求方面,而非修补

漏洞方面。图 9-8 表明了一个大概的维护工作量分布。不同机构中的百分比会有明显不同，但是通常来说，修补系统错误不会是维护活动的主要花费。系统进化过程中要应对新环境和新提出的或者是变更的需求会花费大部分的维护工作量。

维护成本与开发成本的比例在不同应用领域中是不同的。Guimaraes (1983) 的研究表明，对于业务应用系统，维护费用与系统开发成本大体相等。对于嵌入式实时系统，维护费用能达到开发成本的四倍以上。这类系统的高可靠性和高性能需求需要模块间紧密连接，因此改变起来特别困难。尽管这些估计已经过去了 25 年还多的时间，但是不同类型的系统工作量的分布方面似乎没有很大的变化。

在设计和实现上敢于投资来减少系统的维护费用总是值得的。在交付系统之后再添加功能费用就高了，因为需要对现有系统有所了解并分析系统变更可能带来的影响。因此，在开发期间任何降低这种分析费用的工作都会减少维护费用。精确的描述、面向对象开发的使用和配置管理等好的软件工程技术都对维护费用降低是有益的。

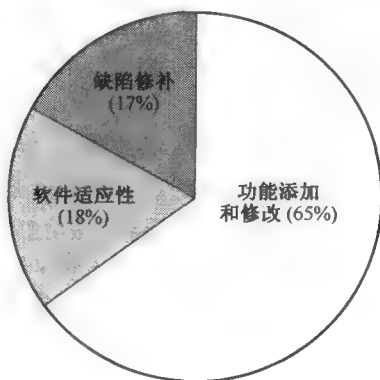


图 9-8 维护工作量分布

图 9-9 说明如果在系统开发阶段多花点儿工夫产生一个可维护的系统，可以使系统整个生命周期成本减少。因为若开发时能充分考虑系统的可维护性，就有在易理解性、分析和测试方面降低成本的可能性，所以会有较高的收益率。系统 1 在开发成本中多投入 25 000 美元，提高了系统的可维护性，结果在整个生命周期中节省了 100 000 美元的维护成本。这表明在开发成本上增加一个百分数，在总的生命周期成本中会下降相应的百分数。

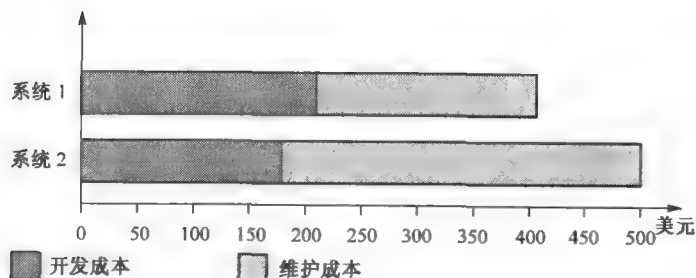


图 9-9 开发和维护的成本



遗留系统

遗留系统是仍能够使用的旧系统，有时对于业务运行来说至关重要。它们可能使用过时的语言或技术实现，或者是使用其他昂贵的系统支持以致维护费用昂贵。通常它们的结构性由于更改、文档的丢失或者已经过时而降低。尽管如此，替换掉这些系统可能是不合算的，因为它们可能仅在一年中的特定时候才被使用。也有可能替换掉它们太过冒险，因为其需求描述已经丢失了。

<http://www.SoftwareEngineering-9.com/Web/Legacysys/>

这些估计虽然是假设的，但是毫无疑问，当整个软件的生命成本被计算进来的时候，开发软件使它更具可维护性是很划算的。这是敏捷开发中重构的依据。没有重构，改变代码会变得越来

越困难和花费更大的代价。然而，在计划驱动的开发中，实际上额外的对提升代码质量的投入在开发过程中是很少做的。这大多取决于机构怎样分配它们的预算。在可维护性方面的投资会导致短期内的成本上升，这是可以度量的。不幸的是，长期的回报不能再同一时间被衡量，所以公司都不情愿对一个未知的未来回报花钱。

通常在系统投入使用之后增加功能，较之在开发期间实现相同的功能代价要高得多。主要原因如下：

1. **团队稳定性** 系统移交之后通常要解散团队，把人员分配到其他新项目中。负责系统维护的新团队或个人既不了解该系统，也不了解系统设计决策的背景，这样在对系统做变更之前就要花费很多精力来理解现有系统。

2. **糟糕的开发实践** 系统的维护合同一般是独立于系统开发合同的。维护合同是与另外的公司签署的，而不是与原开发者签署的。这个因素连同缺乏团队稳定性因素一起，使得开发团队缺乏动力去写维护性好的软件。如果一个开发团队为节约开发量有捷径可走，即使意味着以后软件的改动会更加困难，他们也认为值得去做。

3. **人员技术水平** 维护人员一般都缺乏经验，而且不熟悉应用领域。软件工程人员对维护活动没有什么好印象，通常认为维护不需要太多技术，不如做系统开发那么光彩，所以通常是分配最低级的职员去做。此外，旧的系统可能是用已经淘汰的程序语言写成的，维护人员可能没有多少使用这些语言开发的经验，必须经过学习方能胜任维护工作。



文档

系统文档能够对维护过程有所帮助，它向维护人员提供关于系统结构和组织的信息以及向系统用户提供相关特性。尽管像 XP 这样的敏捷方法的拥护者建议代码应该作为主要的文档，更高层次的关于依赖性和约束的设计模型和信息能够使得其更加容易被理解和更容易对代码进行改动。

<http://www.SoftwareEngineering-9.com/Web/ExtraChaps/Documentation.pdf>

4. **程序年龄和结构** 随着程序不断的变更，其结构受到了破坏。结果是，随着程序年龄的增加，它们变得越来越不容易理解和变更。此外，许多遗留系统没有使用现代化的软件工程技术来开发。这些系统的结构在设计之初就没有规划好，而且系统开发通常只注重效率优化而很少考虑其易理解性。这些老系统的文档要么没有要么就是不完整，还有可能缺乏一致性。旧的系统也没有采用配置管理，因此在进行系统变更时，常常要在寻找系统组件的合适版本上浪费时间。

前 3 个问题的存在是因为很多机构仍然区分系统开发和系统维护。维护被视为第二等的活动，而且没有动力为减少系统变更开销而投资。要想彻底解决这个问题，首先必须接受这样一个观点，那就是系统很少有一个确定的生存周期，而是以某种形态在一个不确定的期限内连续使用。正如前面所说的那样，你应当将系统看成是贯穿于它生命周期的在连续的开发过程中不断进化的。

第 4 个问题，即退化的系统结构问题，在某种程度上讲是最容易解决的问题。可以通过再工程技术来改善系统结构和可理解性。如果适当的话，可以通过体系结构的转换（在本章稍后讨论）使系统适应新的硬件。重构能够提升系统代码的质量并且可以使之更加容易修改。

9.3.1 维护预测

管理者憎恨意外的发生，尤其是那些造成了意想不到的高成本的意外。因此你应当试着去

预测有什么样的系统变更和系统的哪些部分可能是最难维护的。你同样应当试着去估计在给定时限内的系统的总维护成本。图 9-10 说明了对这些不同方面的预测和相关问题。

预测变更请求的数目需要了解系统和外部环境之间的关系。许多系统与外部环境之间存在着复杂的关系，对环境所做的改变不可避免地导致系统变更的发生。要对系统和系统的环境之间的关系做出判断，应该评估以下几点：

1. **系统接口的数目和复杂性** 接口越多、越复杂，接口变更请求就越有可能作为新的需求被提出来。
2. **固有的易变性系统需求的数目** 如第 4 章所讨论的，那些反映机构政策和流程的需求，比那些基于稳定领域特性的需求更容易变动。
3. **系统所处的业务过程** 业务过程在进化的时候，就会产生系统变更请求。使用系统的业务过程越多，要求系统变更的请求就会越多。

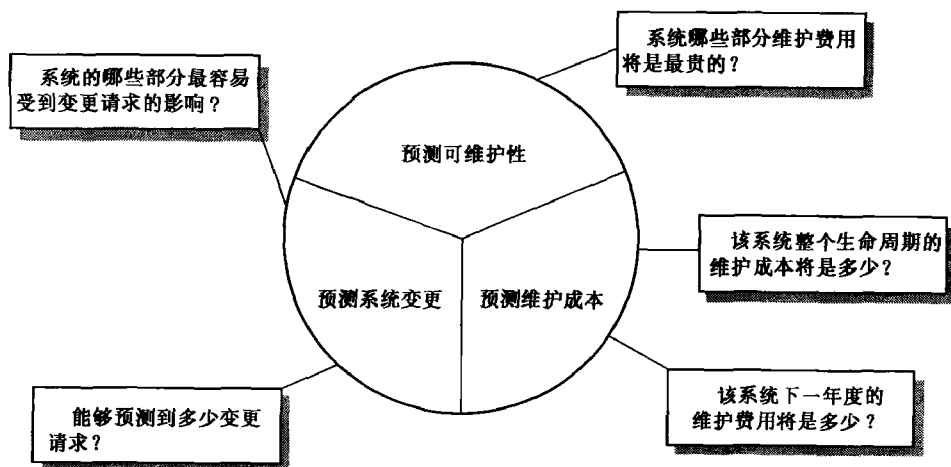


图 9-10 维护预测

很多年以来，研究人员一直在研究程序复杂性和可维护性（Banker 等，1993；Coleman 等，1994；Kafura 和 Reddy，1987；Kozlove 等，2008）之间的关系，程序的复杂性可通过环路复杂度这样的量度衡量（McCabe，1976）。这些研究结论都在意料之中：系统或组件越复杂，其维护费用就越高。复杂性度量在识别那些维护费用特别高的个别程序组件时特别有用。Kafura 和 Reddy 的研究（1987）考查了许多系统组件，发现维护工作量基本集中在少数几个复杂组件上。因此，用比较简单的组件去代替特别复杂的系统组件是划算的。

当系统已经投入服务的时候，可以使用过程数据来帮助预测可维护性。对可维护性评估有用的过程量度如下：

1. **请求纠正性维护的数目** 如果失败报告的数目在增加，这可能暗示着在维护过程期间有更多的错误引入程序之中了，这样可能会导致系统可维护性的下降。
2. **作用分析所需的平均时间** 它反映了受到变更请求影响的程序组件数目。如果这个时间在增加，就暗示着越来越多的组件受到影响，可维护性正在下降。
3. **实现一个变更请求的平均时间** 这不同于作用分析的时间，尽管它们之间可能存在关联性。它所涉及的活动是对系统及其文档进行变更，而不是只评估哪些组件受到影响。变更时间取决于程序设计的难易，因为需要达到性能等非功能需求。如果实现变更的时间在增加，这可能预示着可维护性在下降。
4. **突出的变更请求的数目** 如果这个数目随着时间在增加，它可能意味着可维护性在下降。

我们根据变更请求的预测信息和系统可维护性的预测信息来预测维护费用。多数管理者是将这些信息和自己的直觉、经验相结合来进行成本估计的。成本估计的 COCOMO 2 模型 (Boehm 等, 2000) 指出: 软件维护工作量是基于理解现有代码的工作量和开发新代码的工作量来估计的。详细讨论见第 24 章。

9.3.2 软件再工程

正如在前面章节讨论的, 系统进化过程包括去理解要变更的程序, 然后去实现这些变更。但是, 对于很多系统, 特别是遗留的老系统, 它们是很难理解和进行变更的。这些程序可能最初牺牲了一些可理解性来换取在性能上或空间利用上的改善, 另外, 随着时间的推移, 最初的程序结构经过一系列的变更后已被破坏了。

为了使得遗留系统的维护问题变得更简单, 你可以再工程这些系统以增强它们的结构性和可理解性。再工程包括对系统重新建立文档、重构系统体系结构、用一种更先进的程序设计语言转换系统、修改和更新系统的数据结构和系统的数据取值。一般来讲, 软件的功能不会改变, 也应当避免对系统体系架构的大改动。

再工程相对于直接替换系统来说, 有两个重要的优势:

1. **较小的风险** 对某个关键业务软件的重新开发是要冒很高风险的。系统描述中会发生错误, 而且开发过程中也会出现种种问题。在引入新软件上时间的拖延将意味着商业上的损失且招致额外的花费。

2. **较小的成本** 再工程的成本较之重新开发一个软件的成本来说要小的多。Ulrich (1990) 引用了一个商业系统作为例子, 该系统再实现的成本高达 5000 万美元。最后系统仅用 1200 万美元就成功地实现了再工程。如果运用先进的软件技术, 重新实现的花费与上面的相比可能要少, 但是可以肯定的是它仍然会超过再工程的花费。

图 9-11 是一个再工程过程的通用模型。过程的输入是遗留的程序, 输出是同一个程序的一个已改进和重新构造的版本。这个再工程过程中的活动如下:

1. **源代码转换** 使用转换工具, 将程序从旧的程序设计语言转换到相同语言的一个比较新的版本或另一种语言。

2. **反向工程** 对程序进行分析并从中抽取信息来记录它的组织结构和功能。这个过程通常是全自动完成的。

3. **程序结构改善** 对程序的控制结构进行分析和修改, 使它更容易读和理解。

4. **程序模块化** 程序的相关部分被收集在一起, 在一定程度上消除冗余。在某些情况下,

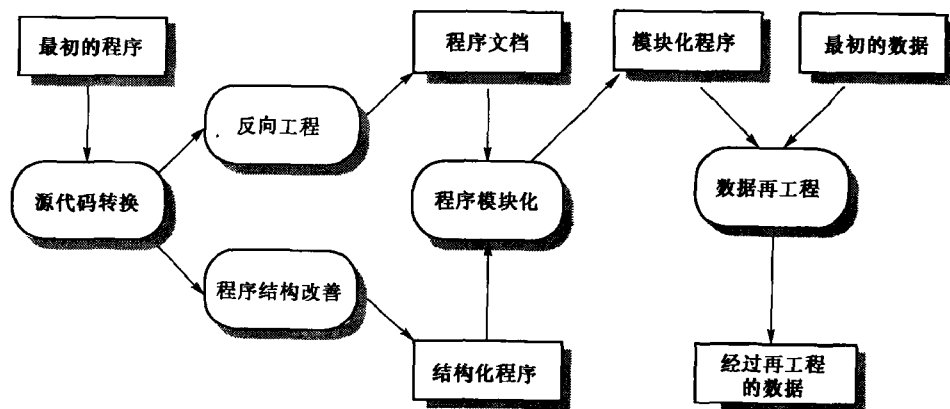


图 9-11 再工程过程

这个阶段可能包括体系结构的重构（例如，一个系统本来使用几个不同的数据存储器，结果被要求使用一个单独的存储器）。

5. **数据再工程** 改变程序处理的数据以反映程序变更。这可能意味着重新定义数据库模式和将已存在的数据库向新的结构转变。你也需要经常清理数据。这包括查找和改正错误、删除冗余记录，等等。通过对工具的使用来支持数据再工程。

程序再工程可能不需要图 9-11 中的所有步骤，如果仍使用应用程序的编程语言，源代码就没有必要做转换；如果再工程完全依赖于自动化工具，那么通过反向工程来恢复文档就没有必要了。数据再工程只有在系统再工程期间程序中数据结构发生了改变时才是需要的。

为了使再工程系统和新软件互操作，我们可能需要开发适配器组件，正如第 19 章所讨论的。这样就隐藏了软件系统的最初的接口，呈现出新的、较好的结构化接口，可以被其他组件使用。对于开发大型的可复用组件来说，遗留系统的封装是一种很重要的技术。

再工程的成本很显然依赖于所做的工作的程度。图 9-12 给出了再工程所可能使用方法的一个谱系。成本从左到右逐渐增长，所以源代码转换是最便宜的选项，而再工程加上一部分体系结构迁移是费用最高的选项。

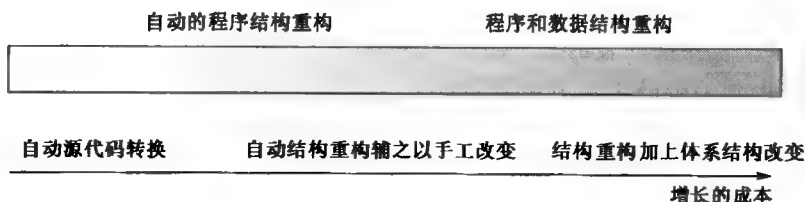


图 9-12 再工程方法

软件再工程的缺点是系统经过再工程能改善的范围受到一定的限制。举例来说，它不可能把面向功能的系统转换为面向对象的系统；主要体系结构的变更或对系统数据管理的重新组织不能自动地执行，因此需要额外的高成本；虽然再工程能改善可维护性，但经过再工程的系统不可能像用现代软件工程方法开发的新系统一样好维护。

9.3.3 通过重构进行预防性维护

重构是提升程序以减缓其由于更改而退化的过程（Opdyke 和 Johnson, 1990）。它意味着通过修改程序来改进程序结构性，降低程序复杂性，让程序变得更加易于理解。重构有时被认为局限于面向对象的开发，但是其原理可以被任何开发方法所使用。当重构一个程序时，不应该增加其功能，而应该注重程序的改进。因此，可以把重构看成是“预防性的维护”，以此来减少将来修改产生的问题。

尽管再工程和重构都是要将软件变得更加容易理解和修改，但它们并不是同一回事。再工程发生在系统已经维护了一段时间并且维护费用不断上升的情况下。通过使用自动化工具来处理并再工程一个遗留的系统，产生一个更具可维护性的新系统。重构是一个连续不断的改进过程，它贯穿于开发和进化的整个过程。重构是要避免导致成本上升和维护困难的结构以及代码的退化问题。

重构是类似于极限编程这样的敏捷方法的一个固有部分，因为这些方法都是应对程序变更的。因此，程序的质量容易退化得很快。所以敏捷开发者经常重构它们的程序来避免这样的退化。对于敏捷方法中对回归测试的强调，降低了由于重构引进新错误的风险。引入的任何错误都会是可检测的，因为之前成功的测试这时会失败。然而，重构不依赖于其他的“敏捷活动”，并

且能够被任何方法用于开发。

Fowler 等 (1999) 表示, 存在一些固定的情况 (他称之为“坏味道”), 程序的代码能够被改进。这些能够通过重构被改进的情况包括:

1. **冗余代码** 在程序的不同地方相似的代码可能重复出现很多次。这种情况可以通过删除和用一个方法或功能去实现。
2. **长方法** 如果方法太长了, 那么这个方法应当被重新设计成几个较短的方法。
3. **选择语句** 这种情况常常牵扯到重复, 因为选择语句 switch 依靠的是同一个值的不同类型。选择语句可能分散在程序的各个地方。在面向对象语言中, 常常可以通过多态性来实现同一个事情。
4. **数据聚集** 当同样的一组数据项 (类中的域, 方法中的参数) 在程序的不同地方重复出现时, 数据聚集就出现了。这通常可以通过用一个对象封装所有数据来解决。
5. **假设的一般性** 这种情况发生在, 当开发者为了以后万一使用到在程序中所包含的一般性。这通常可以简单地删除掉。

Fowler 在他的书和网站中, 也给出了一些基本的重构转换, 这些重构可以被单独或一起使用来处理坏味道。这些转换的例子包括: Extract 方法, 删掉重复的地方并创建一个新方法; Consolidate, 合并条件表达式, 用一个测试替换一系列测试; Pull up 方法, 用父类中的一个方法将各个子类中的类似方法替换掉。交互式开发环境, 例如 Eclipse, 在它的编辑器中包含了对重构的支持。这样使得发现程序中的相互依赖的部分更为容易, 这些部分在实现重构时需要修改。

重构在程序开发期间实现, 是一项有效的降低程序长期维护成本的途径。然而, 当你需要维护一个其结构已经明显退化的程序时, 仅仅通过重构代码是远远不够的。你可能还要考虑设计重构, 这可能是一个花费更高和更加困难的问题。设计重构包括识别相关设计模式 (详见第 7 章) 和用实现这些实际模式的代码替换原先的代码 (Kerievsky, 2004)。这里不做详细讨论。

9.4 遗留系统管理

对于那些新系统, 那些用像增量开发和 CBSE 这样的先进的软件工程过程来开发的新软件系统来说, 就可以规划如何把系统开发和进化很好地结合在一起。越来越多的公司开始认识到系统开发过程是一个全生命周期的过程, 人为地分离软件开发和软件维护是没有好处的。但是, 仍然存在许多遗留系统, 它们是十分重要的业务系统。我们必须扩展和调整它们以适应变化中的电子商务环境。

大多数机构拥有很多遗留系统, 对这些遗留系统的维护和更新资金又非常有限, 这时候就需要对投资做精心的安排, 以期获得最佳的回报。这就要求机构先对遗留系统做现实的评估, 然后做出适当的决策, 有以下 4 种基本选择:

1. **彻底抛弃这个系统** 当系统不能对业务过程产生有效的作用时, 应该选择这个方案。这种情况一般发生在系统安装之后, 业务过程已经改变, 新的业务过程不再依赖于遗留系统。
2. **继续维护这个系统** 当一个系统仍然有存在的必要, 系统运行相当平稳, 而且用户没有提出太多对系统变更的要求时, 应该选择这个方案。
3. **对系统再工程以改善其可维护性** 当系统质量由于经常性的变更已经下降, 而且仍然需要做经常性的变更时, 应该选择这个方案。这个过程应当包括开发新的组件接口, 从而使最初的系统能和其他的新系统协同工作。
4. **以一个新的系统代替整个或部分系统** 当其他因素 (如新的硬件已经使旧系统无法继续

运行, 或者有现成的产品可以使用) 使新系统的开发成本非常合理时, 就应做出此种选择。在很多情况下, 我们可以采用进化替换策略, 用现有的系统替换主要的系统组件, 并且在可能的情况下继续使用其他组件。

当然这些选择不是互斥的, 当一个系统包含多个程序时, 可以根据系统各个部分的实际情况酌情做出不同的选择。

在评估一个遗留系统的时候, 必须从两个不同的视点来看它 (Warren, 1998)。从业务视点看, 必须对该系统的业务价值做出评估。从系统视点看, 必须对应用软件、系统支持软件和硬件质量做出评估。根据这两个方面得到的评估结果就能决定应该对遗留系统采取何种策略了。

例如, 我们假设某机构有 10 个遗留系统。首先评估每个系统的质量和业务价值, 然后画在一张图中, 比较系统在这两方面的相对水平, 如图 9-13 所示。

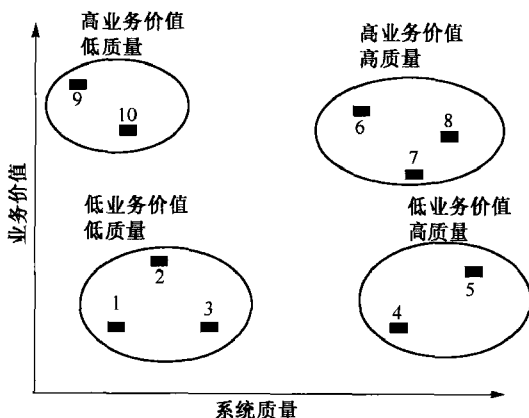


图 9-13 遗留系统评估的一个例子

从图 9-13 中可以看出系统的 4 组分布：

1. **低质量、低业务价值** 保持这些系统继续运转费用很高、回报率很低。这类系统是抛弃的候选对象。

2. **低质量、高业务价值** 这些系统正在为业务做出重要贡献, 不能抛弃。不过, 其低质量意味着运行的成本很高, 所以这类系统有待于转换或者以合适的系统替代。

3. **高质量、低业务价值** 这些系统对业务的贡献很小, 但是维护费用较低。不值得冒险去替换这种系统, 可以继续进行一般的系统维护, 也可以抛弃。

4. **高质量、高业务价值** 这种系统必须保持运转, 其高质量说明无需对其投资进行转换或更换。正常的系统维护应该继续进行。

为了评估一个系统的业务价值, 我们必须看谁是系统的所有者, 比如最终用户和他们的经理, 对系统提出一系列的问题。要讨论的是以下 4 个方面的问题：

1. **系统的使用** 如果系统只是偶尔被使用或被很少一部分人使用, 那么它们含有较低的业务价值。遗留系统可能是为满足某些业务需要而开发的, 但现在业务改变了, 或者有其他更有效的方法满足要求。然而, 一定要注意不经常的但重要的系统使用。例如, 在大学里, 一个学生的注册系统可能仅是在每学年的开始时才使用。但是, 它却是具有高业务价值的必不可少的系统。

2. **支持的业务流程** 当引入一个系统时, 我们就要设计业务流程来开发系统的能力。如果遗留系统难以改变, 那么改变这些业务流程就是不可能的。但是, 当环境变化时, 原来的业务流程已经变得不可用。因而, 由于不能引入新的流程, 而可能使一个系统的业务价值降低。

3. **系统的可靠性** 系统可靠性不仅仅是一个技术问题, 也是一个业务问题。如果系统不可

靠，而且问题直接影响商业用户，或者让业务处理中的人从别的业务转过来解决这些问题，那么系统的业务价值较低。

4. 系统的输出 这里的关键是系统输出对于成功的业务功能的重要性。如果业务依赖于这些输出，那么系统就含有高的业务价值。相反地，如果这些输出可以用某种方法很容易地产生，或者系统产生的输出很少被使用，那么它的业务价值就很低。

例如，我们假设有一家公司提供一个旅行预定系统，负责安排旅行的人员可以通过此系统向认可的旅行代理人发出订单，该公司然后得到预订凭证和发票。但是，业务价值评估可能显示在总的旅行订单中只有相当小的一部分使用了此系统。负责旅行安排的人们发现，通过旅行提供商的网站直接与它们联系更便宜也更方便。这个系统还将继续使用，但是保留这个系统没有什么太大的意义。可以从外部系统得到相同的功能。

相反，如果一个公司开发了一个用来跟踪所有客户以前的订单记录并能自动生成提醒客户续订货物的系统。其结果是有大量的续订订单，而且使得客户倍感满意，因为他们感到提供商了解他们的需要。这样的系统的输出对于业务来说是非常重要的，因此该系统有很高的业务价值。

若从技术的角度来评估软件系统，你要考虑应用系统自身和系统的运行环境。运行环境包括硬件和相关的支持软件（编译器、开发环境等）。环境很重要，是因为很多系统变更是环境改变的结果，如硬件或操作系统的升级。

如果可能的话，在对环境评估的过程中，应该给出系统的度量和系统的维护过程。有用数据的例子包括维护系统硬件和支持软件的花费，在某个确定时间内的硬件缺陷发生的次数，对系统支持软件打补丁和修改发生的频度等。

在环境评估中需要考虑的因素如图 9-14 所示。注意这些因素并不是环境的所有技术特性，还要考虑硬件和支持软件提供商的可依赖性。如果这些提供商不再从事此项业务，他们就不会再提供对系统维护的支持。

因 素	问 题
供应商稳定性	供应商是否还存在？供应商在经济上是否稳定，是否能继续存活？如果供应商已经退出经营，系统是由其他公司维护的吗
失败率	系统是否据传说有很高的失败率？其支持软件是否会崩溃并必须重启系统
年限	硬件和软件的年限有多长？硬件和支持软件越老，就越容易被废弃。虽然它还能正确地工作，但是转移到新系统将是相当经济和有效益
性能	系统的性能充分吗？性能问题对系统用户有严重影响吗
支持的需求	硬件和软件需要哪些局部支持？如果这样的支持是需要高成本的，这时可能就值得考虑做系统替换了
维护成本	硬件维护和软件许可的成本有多少？越老的硬件相比现代系统的维护成本就会越高，软件会有较高的年许可成本
互操作性	系统在与其它系统接口方面有困难吗？编译器等能在目前的操作系统版本上使用吗？需要硬件仿真吗

图 9-14 环境评估中考虑的因素

为了评估应用系统的技术质量，我们要评估一系列因素（见图 9-15），这些因素主要有关于系统的可靠性、维护系统的困难以及系统的文档建立。我们还要收集量化的系统数据来帮助我们对系统质量做出判断。质量评估中可能用到的数据有：

1. 请求系统变更的数目 系统变更容易造成系统结构的损坏，为进一步变更增加了难度。

这个数值愈高，系统的质量也愈低。

2. **用户界面数目** 这对于基于表格的系统来说是一个重要的因素。在这种系统中，每一张表格都可以看做一个独立的用户界面。界面的数目越多，越容易发生界面的不一致性和冗余性。

3. **系统使用的数据量** 使用的数据量（文件的数目、数据库的规模等）越大，就越有可能出现降低系统质量的数据不一致性。

因 素	问 题
可理解性	对当前系统的源代码理解的困难程度如何？其使用的控制结构复杂程度如何？变量有反映其功能的有意义的名字吗
文档编写	哪些系统文档是有效的？文档是否健全、是否一致以及是否反映了当前的现状
数据	系统有一个清晰的数据模型吗？数据在多大程度上在不同文件间是重复的？系统使用的数据是一致的和最新的吗
性能	系统的性能充分吗？性能问题对用户有严重影响吗
程序语言	能得到所用的程序语言的最新编译器吗？所用的程序语言还用于新系统的开发吗
配置管理	系统所有版本和所有部分都受到配置管理系统的管理吗？对当前系统中的组件的版本都有清晰的描述吗
测试数据	对系统的测试数据存在吗？当有新特征加入到系统中来时所执行的回归测试有记录吗
个人技能	能得到优秀技术人员来维护应用吗？是否只有有限几个人了解系统

图 9-15 应用评估中使用的因素

理论上说，应该根据客观的评估结果做出处理遗留系统的决定。然而在许多情况下，做出的决定并不是客观的，而是基于机构或政治上的考虑。举例来说，如果两个机构合并，其中一个机构在政治上地位显赫，它的系统就会被保留下来，另一个机构的系统显然被丢弃。如果高层管理者做出向新硬件平台迁移的决定，那么应用就需要被替换。如果在某一特定的年度内没有用于系统转换的预算，那么系统维护还要继续下去，尽管这将带来较高的长期成本。

要点

- 软件开发与进化应当是一个集成的、完整的、增量式的过程，它可以用螺旋模型表示。
- 对于定制系统来说，软件维护的费用一般超过了软件开发的费用。
- 软件进化过程是由变更请求驱动的，包括变更影响分析、版本规划和变更实现。
- Lehman 定律，例如变更是持续的观点，描述了许多来自于系统进化长期研究的领悟。
- 软件维护有 3 种基本类型：修改软件中的缺陷，使软件适应不同操作环境，以及向系统中添加或修改功能。
- 软件再工程的目标在于改善系统结构和文档，使其更容易理解和变更。
- 应该对遗留系统的业务价值、应用软件的质量以及应用软件的环境进行评估，然后才能决定是否更换、转换和维护系统。

进一步阅读材料

《Software Maintenance and Evolution: A Roadmap》，这篇文章是一个此方面的前沿研究人员关于软件维护和进化的好而简短的概述。他们所认定的研究问题到目前为止还没有得到解决 (V. Rajlich and K. H. Bennett, Proc. 20th Int. Conf. on Software Engineering, IEEE Press, 2000)。
<http://doi.acm.org/10.1145/336512.336534>.

《Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices》, 这本杰出的著作涵盖了软件维护与进化的基本问题, 也包括遗留系统的迁移。此书以一个从 COBOL 系统转换到基于 Java 的客户机 - 服务器系统的大型案例为基础 (R. C. Seacord, D. Plakosh and G. A. Lewis, Addison-Wesley, 2003)。

《Working Effectively with Legacy Code》, 关于处理遗留系统遇到的问题和困难的可靠实用的建议 (M. Feathers, John Wiley&Sons, 2004)。

练习

- 9.1 在现实环境中使用着的软件系统必须进行变更, 否则就会逐渐失去其效用。这是为什么? 试解释之。
- 9.2 试解释 Lehman 定律的基本原理, 并说明在什么情况下该定律不再适用。
- 9.3 从图 9-4 中, 你会发现影响分析是软件进化过程中的一个重要子过程。利用图示, 请问哪些特性需要在变更影响分析中考虑?
- 9.4 假设你是一家专业开发海洋石油工业软件的软件公司的一名项目负责人, 现在你的任务是要找出哪些因素影响你公司开发的特定系统的可维护性。说明你将如何确定一个计划来分析维护过程, 从中发现适合公司的可维护性度量。
- 9.5 简要描述 3 种不同类型的软件维护。为什么有时候很难区分它们?
- 9.6 哪些是在系统再工程花费中需要考虑的主要因素?
- 9.7 如果遗留系统的评估结果表明该系统属于高质量、高业务价值, 那么在什么情况下这个机构却决定抛弃这个系统?
- 9.8 对遗留系统进化的策略选择是什么? 什么时候你通常会替换整个或部分系统而不是继续进行软件维护?
- 9.9 解释为什么在支持软件上出现的问题会使机构不得不替换遗留系统?
- 9.10 在老板没有明确要求的情况下, 软件工程师有无职业责任来生成可维护性代码?

参考书目

- Arthur, L. J. (1988). *Software Evolution*. New York: John Wiley & Sons.
- Banker, R. D., Datar, S. M., Kemerer, C. F. and Zweig, D. (1993). 'Software Complexity and Maintenance Costs'. *Comm. ACM*, **36** (11), 81-94.
- Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. and Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice Hall.
- Coleman, D., Ash, D., Lowther, B. and Oman, P. (1994). 'Using Metrics to Evaluate Software System Maintainability'. *IEEE Computer*, **27** (8), 44-49.
- Erlikh, L. (2000). 'Leveraging legacy system dollars for E-business'. *IT Professional*, **2** (3), May/June 2000, 17-23.
- Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.
- Guimaraes, T. (1983). 'Managing Application Program Maintenance Expenditures'. *Comm. ACM*, **26** (10), 739-46.
- Hopkins, R. and Jenkins, K. (2008). *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston: IBM Press.
- Kafura, D. and Reddy, G. R. (1987). 'The use of software complexity metrics in software maintenance'. *IEEE Trans. on Software Engineering*, **SE-13** (3), 335-43.
- Kerievsky, J. (2004). *Refactoring to Patterns*. Boston: Addison Wesley.

- Kozlov, D., Koskinen, J., Sakkinen, M. and Markkula, J. (2008). 'Assessing maintainability change over multiple software releases'. *J. of Software Maintenance and Evolution*, **20** (1), 31–58.
- Krogstie, J., Jahr, A. and Sjøberg, D. I. K. (2005). 'A longitudinal study of development and maintenance in Norway: Report from the 2003 investigation'. *Information and Software Technology*, **48** (11), 993–1005.
- Lehman, M. M. (1996). 'Laws of Software Evolution Revisited'. Proc. European Workshop on Software Process Technology (EWSPT'96), Springer-Verlag. 108–24.
- Lehman, M. M. and Belady, L. (1985). *Program Evolution: Processes of Software Change*. London: Academic Press.
- Lehman, M. M., Perry, D. E. and Ramil, J. F. (1998). 'On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution'. Proc. Metrics '98, Bethesda, Maryland: IEEE Computer Society Press. 84–8.
- Lehman, M. M., Ramil, J. F. and Sandler, U. (2001). 'An Approach to Modelling Long-term Growth Trends in Software Systems'. Proc. Int. Conf. on Software Maintenance, Florence, Italy: 219–28.
- Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management*. Reading, Mass.: Addison-Wesley.
- McCabe, T. J. (1976). 'A complexity measure'. *IEEE Trans. on Software Engineering*, **SE-2** (4), 308–20.
- Nosek, J. T. and Palvia, P. (1990). 'Software maintenance management: changes in the last decade'. *Software Maintenance: Research and Practice*, **2** (3), 157–74.
- Opdyke, W. F. and Johnson, R. E. (1990). 'Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems'. 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA '90), Poughkeepsie, New York.
- Poole, C. and Huisman, J. W. (2001). 'Using Extreme Programming in a Maintenance Environment'. *IEEE Software*, **18** (6), 42–50.
- Rajlich, V. T. and Bennett, K. H. (2000). 'A Staged Model for the Software Life Cycle'. *IEEE Computer*, **33** (7), 66–71.
- Sousa, M. J. (1998). 'A Survey on the Software Maintenance Process'. 14th IEEE International Conference on Software Maintenance (ICSM '98), Washington, D.C.: 265–74.
- Ulrich, W. M. (1990). 'The Evolutionary Growth of Software Reengineering and the Decade Ahead'. *American Programmer*, **3** (10), 14–20.
- Warren, I. E. (1998). *The Renaissance of Legacy Systems*. London: Springer.

可依赖性和信息安全性

随着软件系统在规模和复杂度方面不断增长，当前我们所面临的软件工程的最为严重的挑战是确保我们能相信这些系统。为了信任这些系统，我们必须保证它在需要使用时是可用的，且能按预期的那样运行。它必须是安全的，我们的计算机或数据不会受到它的威胁。这意味着系统的可依赖性和信息安全性问题通常比系统的功能细节更为重要。本书这部分为此要向学生和从事软件工程的工程技术人员介绍关于可依赖性和信息安全性这个重要话题。

第 10 章涵盖社会技术系统，虽然表面上看社会技术系统与软件的可依赖性和信息安全性没有太大关系，然而，很多信息安全性和可依赖性失败源自人和机构原因，而我们若要考虑可依赖性和信息安全性的话就不能忽视这些方面。软件工程师必须意识到这一点，不应幻想会有更好的技术和手段能确保我们的系统是完全可以依赖的和信息安全的。

第 11 章介绍可依赖性和信息安全性的基本概念，并解释在构建可依赖系统时所采用的避免、检测和恢复策略的基本原理。第 12 章是对第 4 章的补充，主要讲解需求工程，以及关于导出和定义系统的信息安全和可依赖性需求的一些专门方法。此外，还简要介绍形式化描述的使用，其他相关内容则可以从本书网站上获得。

第 13 章和第 14 章是关于开发可依赖和信息安全系统时的软件工程技术，分别阐述可依赖性工程和信息安全工程。但是它们是有很多共性的，不仅讨论软件体系结构的重要性并且展现了帮助获得可依赖性和信息安全性的设计准则和编程技术，也解释为什么使用冗余和多样性来应对失败和外部攻击是很重要的，介绍日趋重要的软件生存能力和恢复能力的话题，它们允许系统在信息安全性受到威胁的时候依然提供必要的服务。

最后，第 15 章关注的是可依赖性和信息安全性保障，解释系统静态分析和模型检测在系统检验和缺陷检测方面的使用。这些技术都成功地应用在有要求极高的系统工程中。此外，还阐述了测试系统可靠性和信息安全性的特殊方法，并解释了为什么一个可依赖性案例对于说服外部的监管者证明所构建的系统安全性和信息安全是必要的。

社会技术系统

目标

本章的目标是介绍社会技术系统的概念，即一个包含有人、软件和硬件在内的系统，并说明你需要从系统角度考虑信息安全性和可靠性。读完本章，你将了解以下内容：

- 掌握社会技术系统的含义，了解基于计算机的纯技术性系统和社会技术系统两者之间的区别；
- 介绍系统总体特性的概念，例如可靠性、性能、安全性和信息安全性等；
- 掌握获取、开发和运行这些包含在系统的工程过程中的活动；
- 了解软件可靠性和信息安全性不能孤立考虑的原因，以及它们如何受到系统因素（如操作员错误）的影响。

在计算机系统中，软件和硬件是相互依赖的。没有硬件，软件系统就是抽象的，它仅仅是人的知识与理念的一种表示。没有软件，硬件只不过是一套没活力的电子设备。但是，如果你将二者放到一起构成一个系统，你就创造了一个可以进行复杂计算并给环境提供结果的机器。

这说明了系统的基本特性之一——系统，大于其各部分相加之和。系统的一些特性只有在各部分集成并在一起运行时才会展现出来。因此软件工程不是一个孤立的活动，而是一个更一般的系统工程过程中的固有组成部分。软件系统不是独立的系统，更多的是含有人、社会或机构目的更广泛系统的基本成分。

例如，野外气象系统软件控制一个气象站里的工具。它与其他软件系统通信，并且是国内及国际气象预报系统的一部分。和软硬件一样，这些系统包括天气预报过程、操作系统及分析结果的人员。系统还包括依靠它向个人、政府、企业等提供天气预报的机构。这类更广泛的系统有时也被称做社会技术系统。他们同时包含了非技术因素，如人、流程、规章等，以及技术因素，如计算机、软件和其他设备。

社会技术系统十分复杂，很难将它当成一个整体理解。更恰当的方法是将他们层次化，如图 10-1 所示。这些层次组成了社会技术系统栈：

1. **设备层** 这一层由硬件设备组成，其中部分是计算机。
2. **操作系统层** 这一层与硬件层交互，为系统中较高层的软件层提供一套公共工具。
3. **通信和数据管理层** 这一层扩展了操作系统工具，提供允许与更多扩展功能交互的接口，比如访问远程系统，访问系统数据库等。因为这一层通常在应用程序和操作系统之间，因此有时被称为中间件。
4. **应用层** 这一层提供所需的特定应用功能，可能包含有多个不同的应用程序。
5. **业务过程层** 利用软件系统的机构业务过程在这一层定义及制定。

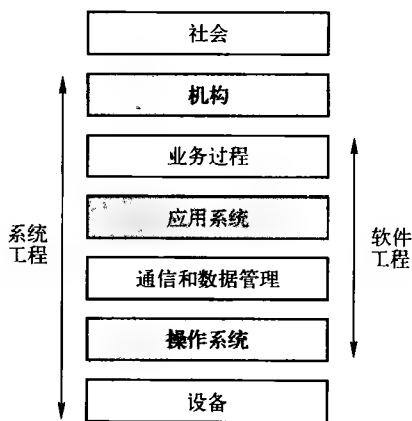


图 10-1 社会技术系统栈

6. 机构层 这一层包含更高层的战略流程以及业务规则、政策和使用系统时要遵循的标准。

7. 社会层 这一层定义了支配系统运作的社会法律和规章。

原则上说,大多数交互是在相邻层进行的,每一层对上一层隐藏下一层的细节。但实际上并不总是这样,有时也存在层次间的意外交互,从而给整个系统带来问题。例如,假设规定个人信息访问的法律发生了变化,这一点属于社会层。它引发了新的机构程序和业务过程的变化。但是,应用程序系统可能不能提供所需层次的细节,因此也要在通信和数据管理层做出相应的调整。

顾全整个系统,而不是简单地从孤立的软件角度来看,这对于考虑软件的信息安全性和可靠性是非常必要的。软件的失败很少会对它自身造成严重的后果,因为软件是无形的存在,即便受到损坏,也可以轻而易举地修复。但是,当这些软件的失败波及系统的其他部分,它们会影响软件的物理的以及人的环境。此时,失败的后果更加严重。人们不得不做额外的工作修补损失,比如设备的损毁,数据的丢失和损坏,或者秘密泄露导致的未知后果。

所以,当你设计一个对信息安全性和可靠性有要求的软件时,你必须从系统全局观察。你必须了解软件失败带给系统中其他部件的后果。此外,系统中的其他部件亦有可能造成、或防止软件失败,以及协助从软件失败中恢复。

因此真正的问题往往是系统的问题,而非软件失败。这说明你需要检查软件与其直接环境的交互方式来确保:

1. 软件失败尽可能包含在系统栈的封闭层中,不会对相邻层的操作产生严重影响。软件失败不应该导致系统失败。

2. 理解系统栈的非软件层错误和失败是如何影响软件的,同时考虑怎样将检测点置入软件中,从而协助检测失败,以及如何提供对失败恢复的支持。

因为软件固有的柔性,许多预想不到的问题同时也就摆在了软件工程师们面前。例如雷达的位置选择以后可能发现雷达图像有扭曲现象,这时通过移动雷达重新定位已经不现实了,解决方案可能就是通过软件来增强雷达图像处理能力,消除扭曲现象。这可能又降低了软件的性能以至于性能变得无法接受。此时问题可能被看成是“软件失败”,而实际上这是系统整体设计过程上的失败。

软件工程师不得不面对在不准许提高硬件费用的前提下增强软件能力的问题,这种情况并不少见。许多所谓的“软件失败”不是软件固有问题的结果。这种失败是尝试通过改变软件以满足已变更的系统工程需求的结果。一个说明这种失败的好例子是丹佛飞机场行李系统的失败(Swartz, 1996),在那儿,遇到了许多所使用的硬件设备上的限制,需要修改控制软件来弥补。

系统工程(Stevens等,1998; Thayer, 2002; Thomé, 1993; White等, 1993)是设计整体系统的过程,不仅是系统中的软件部分。软件是系统中的控制和集成元素,软件工程成本通常是系统成本中的主要成分。作为一个软件工程师,你要了解更多关于软件是如何与其他硬件和软件系统交互的以及希望如何使用它的知识。这些知识能帮助你了解软件的限制,从而设计更好的软件,更好地参与到整个系统工程的团队中来。

10.1 复杂系统

系统这个词用得十分广泛。我们经常谈论计算机系统、操作系统、支付系统、教育系统、政府系统等。显然对“系统”这个词的使用之间存在很大差别,但是它们还是存在一个共同特性,即在一定程度上,系统超出了对它的各个部分的简单堆砌。

抽象的系统,例如政府系统,我们不在这本书中讨论。我这里所讲的系统聚焦在那些包含计算机在内且有某种特殊目的一类系统上,比如能够用于通信、支持导航和工资发放的系统。对这

类系统的一个实用且有指导作用的定义如下：

系统是一组相互关联、不同种类、为实现目标协同工作的组件集合。

这个一般性的定义适合绝大部分系统。举例来说，激光笔是一个简单的系统，它只包括几个硬件组件，外加少量的控制软件。相反，一个空中交通管制系统则是由数千个硬件和软件组件组成的，而且还要有多人参与，他们根据系统信息来做出决定。

所有复杂系统都有一个特征，系统组件的属性和行为会不可避免的交织在一起。每个系统组件机能的成功取决于其他组件的机能。因此，软件只有在处理器运转的情况下进行操作。处理器也只有在定义计算的软件系统成功安装后才可以进行计算。

复杂系统通常具有层次结构并且包含其他系统。比如，警察指挥控制系统会包含一个地理信息系统，以提供事故地点的详细信息。这些被包含的系统称为“子系统”。子系统可以独立地运行，例如，相同的地理信息系统可以用在物流运输和应急指挥与控制系统中。

包含软件的系统可以分为两类：

1. **基于计算机的纯技术系统** 此类系统包括软件组件和硬件组件，但不包括业务过程。这种纯技术性系统的例子有电视、移动电话以及其他嵌入软件的设备。多数个人电脑软件、电脑游戏等也属于此类。个人和机构为特定目的使用纯技术系统，但是有关该目的的知识是不包括在此系统之内的。例如，作者所使用的字处理软件并不知道它正被用来写一本书。

2. **社会技术系统** 此类系统包括一个或多个技术系统，但关键是，系统本身还包括了解系统目的人员。社会技术系统把操作流程和人员（操作者）定义为系统内在的一部分，受制于机构的政策和规则，而且受到外部约束的影响，这些外部因素例如国家法律和调节政策。例如，这本书的产生是通过一种包括了各种流程和技术系统的社会技术出版系统。

社会技术系统是协助提供业务目标的企业系统。此目标可以是提高销售量、减少制造中原材料用量、收税、维护安全空间等。因为系统被嵌入到机构环境中，这些系统的采购、开发和使用会受到机构的政策和流程，以及企业文化的影响。系统的用户则受机构的管理以及与内外部人员的交互方式所影响。

当你尝试开发社会技术系统时，你需要了解它们所使用的机构环境。否则系统可能不符合业务需求，用户以及他们的管理者会否定这个系统。系统环境的机构因素会影响到社会技术系统的需求、设计和运作，它包括：

1. **过程变更** 系统可能对环境中的工作流程有更改需求。如果是这样，则一定存在培训需求。如果是重大变化，或者牵扯到失业问题，那么存在用户抵制采用系统的风险。

2. **工作变更** 新系统可能会降低环境中用户的技术含量或者导致他们的工作方式发生变化。如果是这样，用户会积极抵制机构采用该系统。如需要管理者更改工作方式以适应新计算机系统的设计，这通常会引来不满。管理者会感觉自己在机构中的地位被系统削减了。

3. **机构变更** 系统可能改变机构中的行政权力结构。例如，如果一个机构依赖一个复杂的系统，那么控制该系统访问的人员具有很大的行政权力。

考虑到信息安全性和可靠性问题时，社会技术系统有3个十分重要的特性：

1. **系统整体特性**，即系统作为一个整体而不是各个单个部分所表现的性质。整体特性即依赖于系统组件又依赖于组件之间的关联关系。在此复杂性下，整体特性只能在系统装配完成之后才能评估。信息安全性和可靠性属于整体特性。

2. **系统的非确定性**。这意味着，当给它某个特别的输入时，它不总是产生相同的输出。系统的行为依赖于操作的人，而人不总是以相同的方式反应。此外，对系统的使用可能产生新的系统组件之间的关系，因此改变了总体行为。系统缺陷和失败因此是瞬间的，人们可能在失败是否发生过存在不一样的看法。

3. 系统支持机构目标的程度和范围不仅仅依赖于系统本身。它往往还依赖于这些目标的稳定性, 机构目标之间的关系和冲突, 以及机构中人员是如何解释这些目标的。新的管理人员会重新解释机构当初所设计的系统目标, 由此一个“成功的”系统可能变成一个“失败的”系统。

在决定系统是否成功地满足其目标时, 社会技术性考虑起到关键性的作用。遗憾的是, 对于那些没有什么社会和文化阅历的工程师们来说, 考虑到这些因素对系统的影响是十分困难的。为了帮助理解系统对机构的影响, 人们研究了各种方法, 例如 Mumford 的社会技术学 (1989) 和 Checkland 的软系统方法论 (1981; Checkland 和 Scholes, 1990)。还有大量的有关基于计算机系统对工作影响的社会学研究 (Ackroyd 等, 1992; Anderson 等, 1989; Suchman, 1987)。

10.1.1 系统总体特性

系统中组件之间的复杂关系意味着系统不只是它的各部分的简单的总和, 它还产生一些系统的整体特性。这些“总体特性” (Checkland, 1981) 不能归于任何一个专门的组件部分, 只有从系统整体上看时这些特性才浮现出来。有一些特性, 比如重量, 可以直接从子系统间的特性比较中得出, 但一般情况下则源于复杂的子系统之间的关联关系。系统特性不能直接从单独的系统组件计算得出。图 10-2 所示是一些系统特性的例子。

特 性	描 述
体积	系统的体积 (总的空间占用) 是与组件的装配安排和连接有关的
可靠性	系统的可靠性依赖于组件的可靠性, 但是未预料的交互能引起新的失败类型, 因而影响系统的可靠性
信息安全性	系统的信息安全性 (它的抵御攻击的能力) 是个十分复杂的特性, 很难用尺度来衡量。所设计的攻击可能是系统设计者所无法估计的, 因而也就无法实现内嵌的安全措施
可修复性	此性质反映了当问题一旦发现时修补此问题的难易程度。它依赖于诊断问题的能力, 对有问题组件的访问以及修改和更换这些组件的能力
可用性	此性质反映使用系统的难易程度。它依赖于系统组件的技术、它的使用者以及运行环境

图 10-2 系统总体特性的例子

总体特性有两种类型:

1. 功能特性, 在系统的所有组件集成后系统目的才表现出来。举例来说, 当自行车被装配起来之后就具有了运输工具的功能特性。

2. 非功能特性, 与系统在其运行环境中的行为有关, 如可靠性、性能、安全性和信息安全性。这些特性对基于计算机的系统来说至关重要, 如果在某些特性上达不到最低要求, 系统可能会因此无法使用。某些系统功能可能对某一类用户是不需要的, 因此, 如果该系统不具备这些功能也是可以接受的。然而, 若一个系统是不可靠的或运行速度太慢, 那它就很难被用户接受。

整体可靠性特性, 例如可靠性, 同时取决于单个组件的特性以及它们之间的交互。系统中的组件是相互依赖的。一个组件的失败会传播到整个系统中并影响到其他组件的运作。但是, 预测组件失败对其他组件有怎样的影响通常很困难。因此, 从系统组件的可靠性数据中评估整体系统可靠性是几乎不可能的。

在一个社会技术系统中, 你可以从如下 3 点需要考虑可靠性:

1. **硬件可靠性** 某个硬件组件失效的可能性有多大, 修复那个组件需要多长时间?
2. **软件可靠性** 一个软件组件产生不正确的输出的可能性有多大? 软件失败与硬件失败有一个明显的不同, 那就是软件不存在老化问题。失败通常是暂时的, 软件甚至在出现不正确的结

果之后仍能继续运行。

3. **操作员可靠性** 系统操作员出现操作失误输入错误的可能性有多大? 软件无法检测到错误导致失败传播的可能性有多大?

硬件、软件以及操作员可靠性不是独立的。

图 10-3 表示的就是一个层次的失败如何传播到系统中的其他层次中。硬件失败会产生伪信号, 导致输入超出软件接受范围。软件会有无法预测的行为并产生意外的输出。这会给操作员带来困惑, 使其有压力。

操作员的错误很可能是在其感到紧张的情况下发生的。因此硬件失败可能意味着系统操作员的失误, 从而造成更严重的软件问题或者额外的处理。这会使硬件超负荷, 导致更多的失败等, 因此, 那些本来有可能恢复的初始失败, 会迅速发展成为可能导致整个系统崩溃的严重问题。

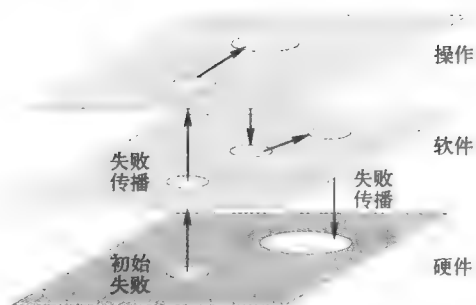


图 10-3 失败传播

系统的可依赖性取决于系统所在的上下文。但是, 系统的环境不能完全定义, 系统设计师也不能为运行系统规定环境上的限制。同一环境下运行的不同的系统会反映出不同的问题, 从而影响整个系统的可依赖性。

例如, 一个系统设定在普通室温下运行。为了允许变化和异常条件下, 系统中的电子元件被设计工作在一个特定的温度范围内, 比如 $0^{\circ} \sim 45^{\circ}$ 范围内。超出此温度范围, 该元件会有异常行为。假设此系统被安装在空调附近, 如果空调失灵, 向系统排放热气, 那么系统会过热。该元件乃至整个系统会出现错误。

如果系统被安装在环境中的其他地方, 这个问题可能不会出现。当空调运行良好时, 没有任何问题。但是, 由于这些机器摆放的很接近, 他们之间的意外联系会造成系统的失败。

像可依赖性一样, 其他的总体特性 (如性能和可用性) 也是很难评估的, 但当系统运行时这些特性是可以测量的。如安全性和信息安全性这样的特性产生的问题是不一样的, 这时你不再只是关心与系统行为相关的属性, 同时也要关心不需要或者不能接受的行为。一个保密的系统是不允许未经授权的人员对数据进行操作的, 但是, 分析出所有可能的进入系统的方式是不可能做到的, 因此, 很难做到对非法闯入的绝对避免。所以对这些“不可以”属性的评估只能是粗略的。只有当真正有人闯入系统时你才知道系统的不安全因素的存在。

10.1.2 系统非确定性

确定性系统是一种完全可预测的系统。如果我们忽略时序问题, 一个在完全可靠的硬件上运行的软件系统, 对于一串输入序列, 总会产生相同的输出序列。当然, 不可能存在完全可靠的硬件, 但是硬件通常足够可靠, 可以认为它是确定的。

在另一方面, 人员是不确定的。提供完全相同的输入 (比如要求完成一个任务), 他们的反应取决于他们的情绪、身体状况、指派任务的人、环境中的其他人以及他们手头上的其他事。有时候他们很乐意完成这份工作, 有时则会拒绝。

社会技术系统是非确定的, 一方面因为它包含人的因素; 另一方面在这些系统中硬件、软件和数据变更的频率太高。这些变更间的交互十分复杂, 因此系统的行为无法预测。这本身并不是一个问题, 但从可靠性角度来说, 它会导致难以确定系统失败是否发生或难以评估系统失败的频率。

例如, 给一个系统提供 20 个测试输入, 处理这些输入的同时记录结果。一段时间过后, 处

理相同的20个测试输入，然后将其结果与前一次记录的结果进行对比，发现有5个不同。这是否说明存在5处失败？抑或这些区别仅仅是系统行为的合理变化？只有在更深入地查看结果并分析系统处理每个输入的方式之后才能得出结论。

10.1.3 成功标准

通常，开发复杂社会技术系统是来解决一些“极复杂问题”（Rittel 和 Webber, 1973）。所谓“极复杂问题”，是指这类问题非常复杂，具有非常多的关联实体，并且无法给出问题的确切描述。不同的信息持有者会从不同的角度来看待这个问题，没有人对此有全面的认识。只有当解决办法有了之后这类问题的本质才暴露出来。这种问题的一个极端例子是地震预报，没有人能准确预报地震的震源在哪儿、什么时间会发生、对局部环境有什么样的影响等。因此，无法详细准确预报强震。

这使得定义系统成功的标准变得很困难。如何确定一个新系统是否能像原先计划的那样提供公司的业务目标的支持？对于是否成功的判断通常不是参照采购和开发系统的最初理由的，而是看它是否在部署的当时是有效。由于业务环境会很快发生变化，业务目标也会在系统开发过程中发生重大变化。

当目标在不同信息持有者的解释下产生冲突，情况会变得更为复杂。例如，MHC-PMS 系统（第1章中讨论过）是基于两个不同的业务目标的：

1. 提高对心理疾病患者护理的质量。
2. 通过提供详细的关于护理和花费的报告以增加收入。

不幸的是，上述两点目标是冲突的。因为第二点目标所需的信息意味着医生和护士必须提供额外的信息，远超过普通健康记录的维护量。这意味着诊所的工作人员陪病患谈话的时间减少因而降低了照顾患者的质量。从医生的角度来说，系统并未较过去的人工系统有所提高；但从管理者的角度来说，的确有所提高。

信息安全性和可靠性的属性的本质有时使得确定系统是否成功变得更为困难。例如，某个新系统的目的是用更安全的数据环境替换已有系统以提高信息安全性。安装后，系统受到攻击，出现安全缺口，一些数据遭到破坏。这是否说明系统是失败的？我们不能确定，因为我们不知道原有系统在受到相同程度攻击时的损失值。

10.2 系统工程

系统工程包括采购、描述、设计、实现、有效性验证、部署、运作和维护社会技术系统的一系列活动。系统工程师不仅要关心系统的软件，还要注意硬件以及系统与用户和周围环境间的交互。它们一定要考虑系统能提供什么样的服务、系统的建设和运行受到什么样的限制、系统为达到其目的是如何使用的。

如图 10-4 所示，大型复杂社会技术系统的生命周期中包括 3 个交叠的阶段：

1. **采购和获得** 在此阶段，系统的目标是确定的，高层系统需求已经建立，已做出关于功能在硬件、软件和人员上的分布分配的决策。采购构成系统的组件。
2. **开发** 系统在此阶段进行开发。开发过程包括与系统开发相关的所有活动，比如需求定义、系统设计、硬件及软件工程、系统集成和测试。此阶段定义运作流程并为系统用户设计培训课程。
3. **运行** 系统在此阶段进行部署，用户完成培训，系统投入使用。规划的运作流程通常会根据系统实际的工作环境进行调整。随着时间的推移，会确定新的需求，系统也会因此进行进化。最终，系统的价值会逐渐下降，直至其退役并被替代。

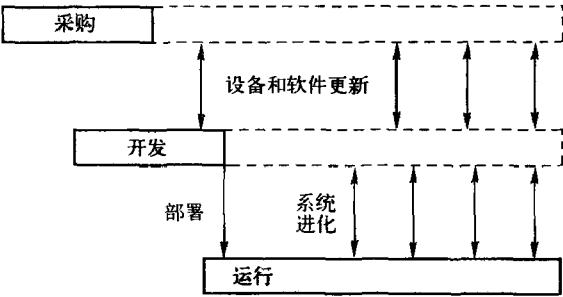


图 10-4 系统工程的阶段

这些阶段并不是独立的。一旦系统进入运行阶段，就要购置新的设备和软件来代替旧系统组件，以提供新功能，或者应对日趋增长的需求。类似地，运作时的变更需求同样需要进一步的系统开发。

系统的总体信息安全性和可靠性受这些阶段的所有活动影响。设计方案会受到系统范围以及系统软硬件的采购决定的限制。由于某些系统安全措施可能无法实现，可能引入一些缺陷，从而导致系统失败。在描述、设计和开发阶段出现的人为错误也会向系统引入缺陷。不全面的测试意味着在部署前没有完全发现系统缺陷。在运行中，部署时的系统配置错误会导致深层的缺陷，系统操作员也会在使用系统时出错。在进行变更时，最初采购做出的假设会被忽略，同样的，这会向系统引入缺陷。

系统工程和软件工程的重要区别在于其整个生命周期内多专业学科的融合。例如，图 10-5 所示的是航空管制系统的采购和开发过程中可能涉及的学科。由于新的航空管理系统通常需要建造新建筑物，系统会需要建筑师和土木工程师。电机和机械工程师会指定和维护电力及空调系统。电子工程师则负责计算机、雷达和其他设备。人体学工程师设计控制工作站，而软件工程师和用户界面设计师负责系统的软件。

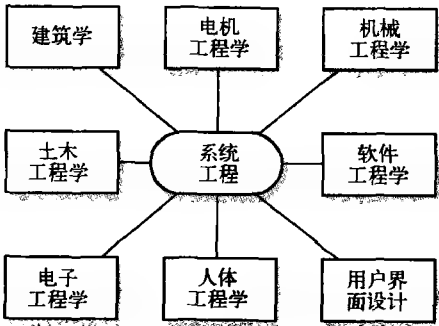


图 10-5 系统工程包含的专业学科

由于复杂社会技术系统包含众多方面，多专业学科的融合变得十分关键。但是，学科之间的区别也会给系统带来缺陷，因此需要对正在开发的系统进行信息安全性与可靠性的折中。

1. 不同学科用相同的词表示不同的含义。不同背景的工程师在讨论中经常出现误解。如果这个问题没有在系统的开发阶段被发现并得到解决，则会导致交付的系统中出现错误。例如，一个略懂一点 C# 编程的电子工程师并不知道 Java 中的方法相当于 C 语言中的函数。
2. 每个学科会被其他学科的人员假定什么可以或不可以做到。这些通常是由于学科间对事实上什么是可能的理解不全面造成的。比如，一个用户界面设计师为一个嵌入式系统构建了一个图形界面，界面对处理的需求量很大，因而造成系统处理器的过载。
3. 科学家们会保护他们的学术领域，他们会争取参与某些设计决断，因为这些设计决定会需要他们的知识。因此，当为某建筑中的门禁系统选型时，尽管机械锁系统可能更可靠，软件工程师还是会争取使用基于软件的门锁系统。

10.3 系统采购

系统工程的最初阶段是系统采购（有时称为系统获得）。在此阶段会决定系统的采购范围、

预算、时间表和高层系统需求。利用这些信息，可以进一步决定是否要采购一个系统、需要何种类型的系统以及找哪个或哪些系统供应商。驱动做出这些决定的力量是：

1. **机构中其他系统的情况** 如果该机构中各种系统混合在一起，彼此间很难通信或维修费用昂贵，那么采购一个代替的系统会给业务带来很大的益处。

2. **遵守外部规则的需要** 业务已经受到越来越多的监管，必须服从外界的规定（例如，在美国要遵循萨班斯法案）。相应的就会对不符合规定的系统有替换需求，或者需要对新系统有特定的是否遵循规定的监控。

3. **外部竞争** 如果一个业务需要有效的竞争或者保持竞争优势，那么对为提高业务过程效率的新系统投资是明智的。比如军事系统中，采购新系统的原因可能是面临新威胁。

4. **业务重组** 业务和其他机构间频繁重组，以提高效率及客户服务。重组会对业务过程造成改变，从而对新系统支持产生需求。

5. **可用的预算** 当前可用的预算决定所采购新系统范围的一个明显因素。

此外，通常会采购新政府系统来表现政治变革和政治策略。例如，一些政客可能决定购买新的监管系统来反击恐怖主义，通过购买此类系统向选民表明他们已经采取行动。但是，此类系统的采购通常没有进行成本效益分析，因而没有比较不同的消费选择所带来的益处。

大型复杂系统通常是现成组件和定制组件的复合体。系统中存在的软件数量越来越多的原因之一是会更更多地用到硬件组件，软件起到“胶水”的作用，使这些硬件组件有效地集成到一起。使用现成组件所节约的成本不如预期的原因之一就是开发这类“粘接剂”的开销。

图10-6所示是一个简单的采购流程模型，针对COTS系统组件以及需要特别设计开发的系统组件。图中所示流程的重点是：

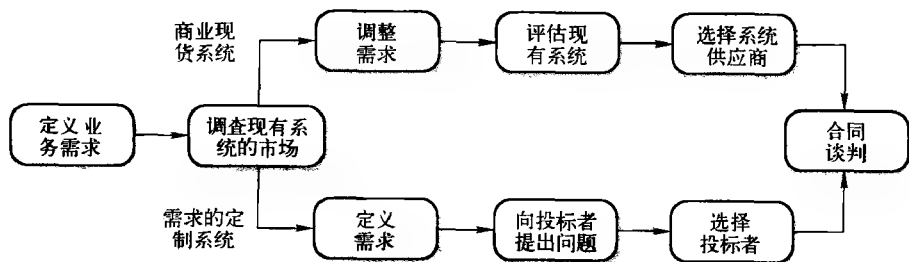


图 10-6 系统采购流程

1. 现成产品组件通常不能完全地满足需求，除非书写需求时已经考虑到了将采用这些组件。因此，选择一个系统就是最大限度地找到系统需求和现成产品组件之间的结合点。

2. 当一个系统要专门开发时，需求的描述是系统获得合同的基础。因此，它既是有法律效力的文件，同时也是技术文件。

3. 选择承包商后，为了开发系统，要通过合同洽谈期，在此阶段协商需求的进一步变更，并商定诸如系统变更的花费等问题。同样的，当COTS系统被选定后，你就需要和提供商就花费、使用许可条件、可能的变更等问题进行洽谈。

社会技术系统中的软件和硬件通常由另外一些机构（提供商）开发，而非采购系统的机构本身开发的。原因在于，采购者的业务很少涉及软件开发，因此其员工不具备自行开发系统所需的能力。事实上，极少有公司具有大型复杂社会技术系统的设计、开发及测试全部组件的能力。

这样的供应商通常被称为总承包商，它可能把不同子系统的开发转包给许多的子承包商。对于大型系统，例如航空交通管制系统，一些供应商会形成一个联盟寻求中标。联盟成员应该覆盖该类系统所需的所有方面，包括计算机硬件供应商、软件开发、外围辅助设备的供应商和一

些特殊设备（如雷达）的供应商。

采购者只与主承包商谈判而不直接与子承包商联系，这样采购者和提供商就只有一个界面。子承包商按照主承包商给出的系统描述来设计和开发。一经完成，这些不同的部分被主承包商整合，然后被交付给采购机构。根据合同，采购者可能允许主承包商自由选择子承包商，或者需要主承包商从一个指定的子承包商列表中选择。

系统采购阶段做出的选择和决定对于系统信息安全性和可靠性有重大的影响。例如，做出一个采购现货系统的决定，这就意味着该机构必须承认他们在信息安全性和可靠性需求方面影响十分有限，而大部分会依赖于系统制造商做出的决定。此外，商业现货系统可能会存在已知的安全隐患或者需要复杂的配置。因配置错误而导致的系统访问端口失去保护，是安全问题的主要来源。

另一方面，采购定制系统的决定意味着要为了了解和明确信息安全性与可靠性需求花费很大的精力。如果一个公司在这方面经验有限，对于他们来说这会是一个非常困难的事情。如果可靠性级别和系统性能要求都要满足，那么开发周期会拖延，开发预算会上升。

10.4 系统开发

系统开发过程的目标是开发或获得系统的所有组件并将其集成在一起从而形成最终的系统。需求是采购和开发阶段之间的桥梁。在采购阶段会定义业务、高层功能性和非功能性系统需求。你可以将此视为开发的起点，迭代的全过程如图 10-4 所示。一旦系统组件的承包商已经确定，就可以进入更详细的需求工程阶段。

图 10-7 是系统开发过程模型。系统开发过程对第 2 章讨论的“瀑布”模型有很大影响。尽管现在已经普遍认为“瀑布模型”模型通常不适用于软件开发，但是多数系统开发过程依然是遵循此模型的计划驱动过程。

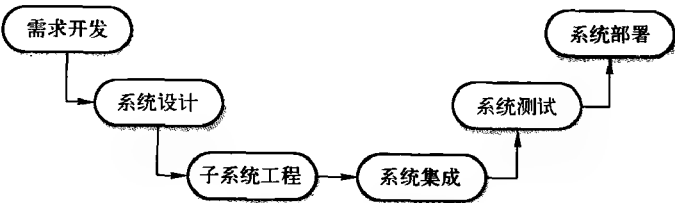


图 10-7 系统开发

在系统工程中使用计划驱动过程的原因是系统的不同部分在同时开发。对于包含硬件和其他设备的系统来说，开发过程中出现的变更成本是很高的，或者说有时实际上是不可能的。因此在硬件开发或创建工作开始前充分理解系统需求是十分关键的。重做系统设计以解决硬件问题的可能性很低。正因为如此，越来越多的系统功能被分配到系统软件中去。这样一来便可以在系统开发阶段进行某些改动，以对应不可避免会有所增长的新系统需求。

系统工程中最容易混乱的问题之一是公司之间对过程中的每个阶段所使用的不同术语，过程的结构也五花八门。有时需求工程是开发过程的一部分，有时则是一个独立的活动。但是，本质上来说系统开发阶段分为如下 6 个基本活动：

- 1. **需求开发** 采购阶段所定义的高层和业务需求要得到进一步的完善。需求要具体到硬件、软件，或过程和实施的优先级。
- 2. **系统设计** 这个过程和需求开发过程有效的迭代。其中包括建立系统总体体系结构、识别系统不同组件以及理解它们之间的关系。

3. **子系统工程** 这个阶段包括软件组件的开发、现成软硬件的配置、设计专用硬件（如果有需要的话）、定义系统运作流程以及重新设计基本业务过程。

4. **系统集成** 此阶段会将所有的组件集成在一起组成新的系统，只有这样系统总体特性才会显现。

5. **系统测试** 这是一个高强度而旷日持久的活动，通常会发现问题。需要重新进入子系统工程和系统集成阶段来修复这些问题，调整系统性能并实现新的需求。系统测试同时涉及系统开发人员和采购系统的机构用户测试。

6. **系统部署** 此时系统进入可用阶段，从已有系统获得数据，并环境中的其他系统建立通信。用户开始使用系统来支持他们工作时过程达到峰值。

尽管整体过程是计划驱动的，但需求开发和系统设计过程是相连的。需求和高层设计是同时发生的。现存系统的约束会限制设计选择，这些选择明确出现在需求描述中。你必须为需求工程过程的结构和组织关系做初始设计。在设计过程当中，你还会发现已有需求的新问题，同时还会有新的需求冒出来。因此，你可以将这两个相互关联的过程看成是一个螺旋，如图10-8所示。

螺旋过程反映了这样的事实，即需求影响设计决策，反过来设计也会作用于需求。所以将这两个过程交织在一起是有意义的。从中央部分开始，螺旋的每一圈都会增加细节内容到需求和设计。有些环集中于需求，而有些环集中在设计。有时，在需求和设计过程中收集了新知识，这意味着问题阐述本身可能需要改变。

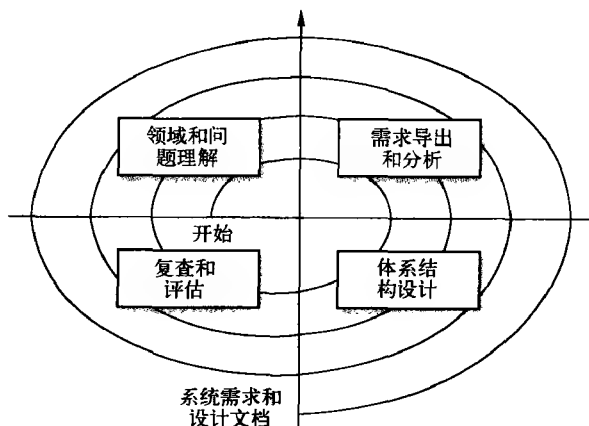


图 10-8 需求和设计螺旋

对于几乎所有的系统，都有许多可能的设计方案，包括硬件因素、软件因素和人的因素的多种组合的选择。有进一步开发余地的解决方案也许是最适当的技术方案。然而，在许多情况下，来自机构和政治上的影响力会对方案的选择产生举足轻重的作用。举例来说，如果系统是为政府定制的，那政府可能较喜欢让本国的供应者而不是国外的厂商去做，即使别的国家的产品技术更好。这些影响总是要在螺旋模型中的复查和评估阶段发生作用，而此时需求和设计就有可能被接受也有可能遭到拒绝。当复查表明需求和高层设计已经足够详细，允许子系统进行描述和设计时，本过程就可以结束了。

在子系统工程阶段，会实现系统的硬件和软件组件。对于一些类型的系统，比如航行器，所有的硬件和软件组件都在开发过程中完成设计和实施。但是对于大多数系统，有些组件属于商业现货系统（COTS）。购买现成的产品通常比开发专用的组件要便宜很多。

不同的子系统通常是并行开发的。当出现了超过子系统范围的问题，就会发生一个系统修改的请求。当系统包括很多昂贵的硬件工程时，制造后再修改的成本往往很高。这时为解决问题就会千方百计寻找变通方法。由于软件的固有柔性，这些变通方法会涉及软件的变更。

系统集成就是将一个个独立开发的子系统集成为一个完整的大系统。集成可以用一种同时将所有的子系统集成在一起的所谓“大爆炸”式的方法。然而，由于技术上和管理上的原因，增量式集成子系统的过程是最恰当的方法，该集成过程分成了多个子过程，每个集成子过程只集成一部分子系统：

1. 不同子系统开发的时间是无法准确预计的，因此，不大可能在同一时间得到所有的子

系统。

2. 增量式集成可以减少错误定位的成本。如果许多子系统同时被集成,测试时发生了错误就需要在这些子系统中定位。若将单个子系统集成到已经能运行的系统中来,一旦发生问题,很明显,所发生的错误或是在新进入的子系统或是在系统和新进入的子系统之间的交互当中。

随着越来越多的子系统是由集成 COTS 硬件和软件组件来构成的,实现和集成之间的区别越来越模糊了。在某些情况下,没必要开发新的软硬件,集成已成为系统实现的最主要工作了。

在集成过程中及之后的时间段中会进行系统测试。测试要侧重组件之间的借口以及系统整体的行为。这会不可避免地暴露单个子系统的问题,必须进行修复。

在系统集成阶段一些子系统故障就会暴露出来了,这些故障主要是源于对其他子系统所做的一些无效假设引起的。这些可能引起子系统的承包商之间的争论。当问题出在子系统之间的交互中时,不同的承包商可能相互推卸对问题的责任。就如何解决问题上的谈判可能需要花上数个星期或数月的时间。

系统开发的最后阶段是系统交付和部署。此时软件已经安装到硬件上,做好一切运行的准备。这会涉及系统针对其使用环境的配置,从已有系统的数据传输以及用户文档和培训的准备。在此阶段,你需要重新配置环境中的其他系统来确保新系统能和它们共同运转。

虽然原则上很简单,但部署阶段会出现很多困难。用户环境会和系统开发者所预期的有所差异,调整系统使之与不同用户环境相适应并非易事。现有数据可能需要大面积清理,部分数据也可能丢失。其他系统的接口可能没有完善的文档。

系统开发过程对可靠性和信息安全性的影响是显而易见的。有关可靠性和安全性的需求以及在成本、进度、性能和可靠性间的取舍决定都是在此阶段做出的。所有阶段出现的人为错误会向系统引入错误,这会在运行中引发系统失败。测试和认证过程不可避免地受到成本和时间的约束。因此,系统可能没有得到全面的测试。用户会在使用系统的过程中完成测试。最终,系统开发的问题会意味着系统和其运行环境有所不符。这些会导致系统使用中的人为错误。

10.5 系统运行

运行过程包括按照系统定义的目的来使用系统。例如航空交通管制系统的操作员在飞机进入或离开领空、变更高度或速度、发生紧急情况时遵循特定的流程。这些操作过程必须在系统开发时进行定义并撰写文档。操作员需要进行培训,其余工作流程要做出调整以有效使用新系统。由于系统需求描述存在问题或冗余,在此阶段会暴露出未检测出来的问题。尽管系统是按照需求描述运行的,但其功能可能和实际操作需求不符。因此,操作员不会像设计者所想那样使用系统。

有系统操作员的主要好处是人员对意料之外的情况有很好的应对能力,即便是当他们从来没有直接经历过这些情况。因此,当系统出现问题,操作员通常可以解决该情况,尽管有时这会导致定义的过程被破坏。操作员还会运用自己的知识来适应和改进过程,通常实际运行过程与系统设计师所预期的过程有所差异。

因此,我们要将运行过程设计得灵活且可调整。运行过程不应该包括太多的约束,也不应该对操作顺序有过多的要求,系统软件不应该依赖于遵循特殊的过程。由于操作员知道实际情况中哪些可以或不可以正常工作,因此他们通常可以改进该过程。

有一个问题只有在系统投入使用后才会出现,就是新系统和现有系统共同工作的问题。可能会是不兼容的问题,也可能是系统间数据传输的困难。不同系统用户界面的区别也可能产生更细微的问题。由于操作员错误地使用另一个系统的用户交互指令,新系统的引入会增加操作员的失误率。

10.5.1 人为错误

前面介绍过，非确定性是社会技术系统的重要问题，其中一个原因是系统中的人员不会总是做出同样的行为。有时他们在使用系统过程中犯错误，这可能引发系统失败。例如，一个操作员忘记记录某些发生的操作，另一个操作员（错误地）重复了这个操作。如果这个操作是借记或银行贷款，那么由于账户总量错误，系统会失败。

正如 Reason 所讨论（2000）的那样，人为错误总会发生，可以从两个角度来看待这个问题：

1. **人的方法** 错误被认为是个人的责任，而“不安全行为”（如操作员没有使用安全防护栏）是个人疏忽或不计后果行为。采用这个视角的人通常认为人为错误可以通过更多的训练、更严格的程序和重新训练内容等来减少。他们认为错误是犯错人的责任。

2. **系统的方法** 这个角度是基于人会犯错误也允许犯错误的假设。系统设计的错误决定会导致错误的工作方式，机构的因素也会影响到系统操作员，这些会引发人员出错。好的系统应该可以识别人为错误，同时包括检测人为错误的保障系统并允许系统在失败发生前进行自我修复。如果发生系统失败，不应寻找个人为此负责，而是要了解系统方面为什么没有能够拦截该错误。

系统角度才是正确的视角，系统工程师要假设系统运作时会发生人为错误。因此，为了提高系统的信息安全性和可靠性，设计师要考虑在系统中加入人为错误的防范措施。同时，他们需要考虑这些防范措施是否应该构建到系统的技术组件中去。如果不需要，那么这些措施要包含在使用系统的过程中，或者作为操作员的指南，依赖于人为的检查和判断。

下面是系统中包含防范措施的几个例子：

1. 航空交通管制系统包含一个自动冲突报警系统。当指挥员指挥一架飞机改变其速度或高度时，系统推断其轨道并检测是否与其他飞机轨道交叉。如果是，响铃警报。

2. 该系统包含一个定义清晰的流程以记录做出的控制指令。指挥员可以通过这些流程来检查自己是否正确下达指令，其他人员亦可检查这些信息。

3. 航空交通指挥通常包括一组自始至终监控其他人工作的指挥员。因此，当发生了一个错误，在事故发生之前就会被发现。

所有的安全保证措施都不可避免地存在某些缺陷。Reason 称其为“潜在条件”，因为它们通常只会当其他问题发生时才引发系统失败。例如，在上述防范措施中，冲突警报系统的缺陷在于它可能会造成很多假警报。指挥员会因此忽略系统的警报。过程系统的缺陷是不能简单地记录那些不常见但很关键的信息。当所有的人都处于紧张状态时会犯同样的错误，此时人为检查会失效。

在系统内防范措施没能有效拦截系统操作员的失误时，潜在条件会导致系统失败。人为错误是失败的触发器，但不应该认为是失败产生的唯一原因。Reason 用其著名的系统失败“瑞士奶酪”模型解释了这一点（见图 10-9）。

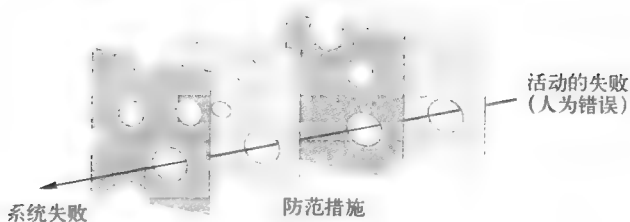


图 10-9 Reason 的系统失败瑞士奶酪模型

这个模型把系统内建的防范措施比作一片瑞士奶酪。有些种类的瑞士奶酪会有很多孔，比如瑞士多孔干酪，类似的，这些潜在条件就是奶酪片上的孔。奶酪上孔的位置不是固定的，而是随着整个社会技术系统的状态而改变的。如果每一片瑞士奶酪代表一层防范措施，当某一位置所有的孔对齐时会出现失败，造成人为操作错误。系统运行的一个主动失败会从孔中穿过并导致整体系统的失败。

当然，通常这些孔并不是对齐的，因此运行失败会被系统拦截。为了减少人为错误引发系统失败的可能性，设计师应该：

1. 设计一个包含多种防范措施的系統。这就意味着“孔”会在分布在不同的位置，从而降低所有的孔对齐、无法拦截错误的可能性。

2. 最小化系统中的潜在条件数，即减少系统“孔”的大小和数目。

系统的整体设计还应该尝试避免触发系统失败的那些主动失败。这可能包括设计运行过程及系统，以确保操作员不会过度工作、分神或被过多的信息所淹没。

10.5.2 系统进化

大型和复杂的系统都会有一个非常长的生存期。在整个生存期内，必须改进原先系统需求中的错误进而满足出现的新需求。系统中的计算机设备更新是有可能的。使用系统的机构可能重新改组并且因此以一不同的方式使用系统。系统的外部环境可能改变，所有这些都会要求系统随之变化。因此，系统为适应环境变化而做出的进化是系统运行过程中的一环。系统进化包括重新进入开发流程做出改变、系统软硬件的扩展及运行过程。

像软件进化一样，系统进化是昂贵的，其原因如下：

1. 必须从业务和技术的角度对提议的变更做仔细的分析。在变更生效之前必须经过有关人员的认同。

2. 因为子系统都不是孤立存在的，对一个子系统的变更可能造成其他子系统的性能或行为的负面影响。因此，对其他子系统的相应变更可能是需要的。

3. 最初设计决策的理由时常未被记录。把一个特别的设计决策产生的原因找出来，这对于系统进化是有意义的。

4. 当系统运行相当长时间以后，其结构被修改得复杂而凌乱，系统进化的成本进一步增加。



遗留系统

遗留系统是以计算机为基础的社会技术系统，它们是在过去开发出来的，往往采用的是今天看来是旧的和不再使用的技术。这些系统不仅包括硬件和软件，还包括遗留的过程和程序，即一些老的做事方法，由于依赖于遗留软件而很难加以改变。对此类系统其中某一部分的改变都会带来对其他组件的改变。遗留系统通常都是关键性业务系统。维护它们的原因就是更换它们需要冒太大的风险。

<http://www.SoftwareEngineering-9.com/LegacySys/>

随时间推移而进化的系统往往是依赖于旧的硬件和软件技术。如果系统在机构中占有至关重要的地位的话，我们就称其为“遗留系统”。这些系统通常是机构想替换的，但是考虑到替换的风险和代价，无法进行替换。

从可靠性和信息安全性的角度来说，系统的变更通常会导致问题和缺陷。如果实施变更人员并非参与系统开发的人员，他们会忽略那些考虑到可靠性和信息安全性的因素而做出设计决

定。因此，他们会变更系统，并且破坏系统构建时所实施的安全屏障。此外，由于测试费用昂贵，不可能在每个变更完成后进行完全的测试，导致变更对系统组件造成或引入的错误不能被发现。

要点

- 社会技术系统是机构内部的包括计算机硬件、软件和人的系统。设计它们是为了支持机构或业务目标。
- 人和机构因素（例如机构结构和管理）对社会技术系统的运行有绝对的影响。
- 系统的整体特性是系统作为一个整体表现出来的特性而不是各个组件部分的特性。这些特性包括性能、可靠性、可用性、安全性和信息安全性等。系统的成功或失败时常依赖于这些系统特性。
- 系统工程的基本过程是系统采购、系统开发和系统运作。
- 系统采购涵盖决定购买什么系统、由谁来提供系统的所有活动。采购流程也包括定义高层需求。
- 系统开发包括需求描述、设计、构建、集成和测试。系统集成将多个供应商提供的子系统集成在一起，因此这个环节十分关键。
- 当系统投入使用，运行过程及系统本身需要做出相应的变更以满足业务需求的变更。
- 人为错误是不可避免的，系统应该包含可以在系统失败前检测到错误的安全防范措施。Reason 的瑞士奶酪模型解释了人为错误与安全屏障的潜在缺陷是如何导致系统失败的。

进一步阅读材料

《Airport 95: Antomated baggage system》，是一本非常棒的而且具有较好可读性的书，全书侧重两个方面：一是对系统工程项目中容易出问题的地方；二是对系统失败所招致的软件责难进行了透析（ACM Software Engineering Notes, 21, March 1996）。<http://doi.acm.org/10.1145/227531.227544>。

《Software system engineering: A tutorial》，这是一篇很好的软件工程综述文章，尽管 Thayer 只关注于基于计算机的系统，并没有讨论社会技术方面的问题（R. H. Thayer, IEEE Computer, April 2002）。<http://dx.doi.org/10.1109/MC.2002.993773>。

《Trust in Technology: A socio-technical Perspective》，这是一本有关社会技术系统的可靠性的论文集（K. Clarke, G. Hardstone, M. Rouncefield and I. Sommerville (eds.), Springer, 2006）。

《Fundamentals of Systems Engineerings》，这是 NASA 系统工程手册的绪论。表述了航天系统的系统工程过程。虽然多数是技术系统，但是这当中依然存在一些社会技术因素需要考虑。可靠性显然十分重要（In NASA Systems Engineering Handbook, NASA-SP2007-6105, 2007）。[http://education.ksc.nasa.gov/esmdspacegrant/Documents/NASA% 20P- 2007- 6105% 201% 20Final% 2031Dec2007. pdf](http://education.ksc.nasa.gov/esmdspacegrant/Documents/NASA%20P-2007-6105%201%20Final%2031Dec2007.pdf)。

练习

- 10.1 举出两个由复杂社会技术系统支持的政府功能，并解释为什么这些功能不能完全自动完成。
- 10.2 解释为什么基于计算机的系统所安装的环境可能对系统有不可预知的影响并导致系统失败。请用不同于本章描述的例子阐述你的观点。
- 10.3 为什么不能从系统组件的特性来推断整个复杂系统的总体特性？
- 10.4 为什么有时很难判断社会技术系统是否发生了系统失败？试用之前讨论过的 MHC-PMS 为例阐述你的观点。

- 10.5 什么是“极复杂问题”？解释为什么开发全国医疗记录系统会出现“极复杂问题”。
- 10.6 欧洲博物馆协会准备开发一套博物馆多媒体系统提供古希腊的虚拟体验。该系统通过一个标准的 Web 浏览器让用户观看三维古希腊模型，也提供沉浸式虚拟现实体验。在博物馆协会内部的各个博物馆安装此系统会遇到哪些政治和机构方面的困难？
- 10.7 为什么说系统集成是系统开发过程的关键环节。举出 3 个可能导致系统集成出现问题的社会技术系统因素。
- 10.8 解释为什么遗留系统对于业务运行是至关重要的。
- 10.9 像电气工程或软件工程一样，把系统工程视为一个独立学科，其赞成的理由和反对的理由各是什么？
- 10.10 设想你是一个参与金融系统开发的工程师。安装过程中，你发现这个系统将会使很多人成为冗员。相关人员抵制你访问必要的信息以完成系统的安装。作为一个系统工程师你应该在多大程度上涉入这件事？按合同中规定的那样完成安装是你的专业责任吗？还是应该简单地放弃工作直到用户单位解决这个问题？

参考书目

- Ackroyd, S., Harper, R., Hughes, J. A. and Shapiro, D. (1992). *Information Technology and Practical Police Work*. Milton Keynes: Open University Press.
- Anderson, R. J., Hughes, J. A. and Sharrock, W. W. (1989). *Working for Profit: The Social Organization of Calculability in an Entrepreneurial Firm*. Aldershot: Avebury.
- Checkland, P. (1981). *Systems Thinking, Systems Practice*. Chichester: John Wiley & Sons.
- Checkland, P. and Scholes, J. (1990). *Soft Systems Methodology in Action*. Chichester: John Wiley & Sons.
- Mumford, E. (1989). 'User Participation in a Changing Environment—Why we need it'. In *Participation in Systems Development*. Knight, K. (ed.). London: Kogan Page.
- Reason, J. (2000). 'Human error: Models and management'. *British Medical J.*, **320** 768–70.
- Rittel, H. and Webber, M. (1973). 'Dilemmas in a General Theory of Planning'. *Policy Sciences*, **4**, 155–69.
- Stevens, R., Brook, P., Jackson, K. and Arnold, S. (1998). *Systems Engineering: Coping with Complexity*. London: Prentice Hall.
- Suchman, L. (1987). *Plans and situated actions: the problem of human-machine communication*. New York: Cambridge University Press.
- Swartz, A. J. (1996). 'Airport 95: Automated Baggage System?' *ACM Software Engineering Notes*, **21** (2), 79–83.
- Thayer, R. H. (2002). 'Software System Engineering: A Tutorial.' *IEEE Computer*, **35** (4), 68–73.
- Thomé, B. (1993). 'Systems Engineering: Principles and Practice of Computer-based Systems Engineering'. Chichester: John Wiley & Sons.
- White, S., Alford, M., Holtzman, J., Kuehl, S., McCay, B., Oliver, D., Owens, D., Tully, C. and Willey, A. (1993). 'Systems Engineering of Computer-Based Systems'. *IEEE Computer*, **26** (11), 54–65.

可依赖性与信息安全性

目标

本章的目标是介绍软件可依赖性和信息安全性。读完本章，你将了解以下内容：

- 理解为什么一个软件系统中可依赖性和信息安全性通常比功能特性更重要；
- 理解可依赖性的 4 个主要方面，也就是可用性、可靠性、安全性和信息安全性；
- 了解在讨论信息安全性和可依赖性问题时所用到的专门术语；
- 理解想要得到一个保密、可靠的软件，就需要在软件开发阶段避免错误，在系统使用当中检测和排除错误，还要限制由于系统失败造成的损失。

随着计算机系统已经深入到我们的业务和个人生活当中，系统与软件的失败所带来的问题也在增加。一个电子商务公司的服务软件的失败有可能导致收入上较大的损失，并有可能殃及这家公司的顾客。一个嵌入到汽车上的控制软件发生错误可能会导致昂贵的召回和维修费用，并且在最坏的情况下，可能成为导致意外事故的因素之一。公司里的电脑如果被恶意代码感染，则需要昂贵的清除操作以找出问题所在并可能导致损失或敏感信息的毁坏。

因为软件密集型系统对于政府、公司和个人都如此重要，所以广泛使用的软件必须是可信的。这种软件应该在需要时是可用的，并且应该能够正确地工作且不会出现不受欢迎的副作用，例如未授权的信息泄露。“可依赖性”（dependability）这个概念是由 Laprie（Laprie, 1995）提出来的，指的是系统的可用性、可靠性、安全性和信息安全性。正如 11.1 节所讨论的，这些属性是交织在一起的，所以用一个词来覆盖它们是有意义的。

以下几点理由说明了系统的可依赖性现在通常比它们的具体功能更加重要：

1. **系统失败影响到的人数众多** 很多系统中包含了一些很少被使用的功能。如果这些功能被从系统中除掉，只有一小部分人会受影响。而影响系统可用性的系统失败潜在地影响所有使用系统的用户。系统失败可能意味着正常的业务无法进行。

2. **如果系统是不可靠的、不安全的或是不保密的，那用户往往会拒绝使用它** 如果用户不信任一个系统，他们就会拒绝使用它。更有甚者，他们还有可能拒绝购买和使用来自同一个公司的其他产品，因为他们认为所有的系统可能都是这样不可信任。

3. **系统失败的代价可能是巨大的** 对于某些应用，例如反应堆控制系统或者飞机导航系统，系统失败的代价要比一般控制系统的失败大好几个数量级。

4. **靠不住的系统能导致信息流失** 数据收集和维持的成本很高，有时甚至比处理它的计算机系统还贵。恢复丢失的或被损坏的数据所需要的花费往往非常高。

正如第 10 章中所讨论的，软件通常是一个大的系统中的一部分。它在一个运行环境下执行，这个环境中包括有能令软件在其上执行的硬件、软件的使用者，还有机构过程或业务过程。因此当设计一个可依赖的系统时，我们需要考虑：

1. **硬件失败** 系统硬件失败有可能是源自设计上的失误，也可能源自组件加工制造中的问题，或者是硬件组件已达到它们的使用年限了。
2. **软件失败** 系统软件问题可能是由于描述、设计和实现中的错误。
3. **操作失败** 系统的操作人员未能正确地使用系统。因为硬件和软件正在逐渐变得可靠，

所以操作上的失败就成为最大的一个引起系统失败的原因了。



要求极高的系统

有些类型的系统是“要求极高的系统”(critical system)，如果系统失败会造成人身伤害：环境破坏，或大规模经济损失。要求极高的系统的例子包括：嵌入在医疗设备中的系统，比如胰岛素泵（安全性要求极高的系统），宇宙飞船的导航系统（任务要求极高的系统），在线转账系统（业务要求极高的系统）。

要求极高的系统开发费用非常昂贵。不只因为它们必须被开发为失败概率非常小的系统，还因为它们必须包含当失败发生的时候需要使用到的恢复机制。

<http://www.SoftwareEngineering-9.com/Web/Dependability/CritSys.html>

这些失败可能是相互关联的。硬件组件的失败可能需要系统操作员不得不处理不可预料的情况，因而增加了额外的负荷，这使得他们精神紧张——人在这种紧张情况下更容易出错。这又会导致软件失败，软件失败又意味着需要操作员做更多的工作，神经更加紧张，等等。

由此看来，对于开发可靠的、软件密集型系统的设计者来说尤其重要的是要有系统观，不能只注意系统的某个局部。如果系统的硬件、软件和操作过程的设计是孤立考虑的，一个部分的设计者没有注意到其他部分的潜在弱点，那么就有可能在系统的各组件接口处出错。

11.1 可依赖性特征

我们都知道计算机系统失败是怎么一回事。在没有明显原因的情况下，计算机系统有时会崩溃或者出错。在计算机上运行的程序没有像预料的那样执行，有时可能损坏系统中的数据。我们已经习惯了这些失败，很少有人完全信任所使用的计算机。

计算机系统的可依赖性是用来衡量其可信度（trustworthiness）的性能指标。可信度表现为用户对系统的信任程度，系统是否能按照他们预期的那样操作以及系统是否会在正常使用中失败。以数值量化可依赖性没有多大意义，相反，我们将其分为这样几个层次：“不可依赖”、“非常可依赖”、“极度可依赖”。

可信赖和有用不是一回事。写这本书所用的字处理器在作者看来不是很可靠，它有时会死机而需要重启。尽管如此，因为它很好用，所以作者可以容忍偶尔的失败。然而作者对这个系统的不信赖可能反映为，作者要很频繁地进行存盘操作并保存文档的多个拷贝。作者通过限制系统失败带来的损失来弥补系统的可依赖性上的不足。

如图 11-1 所示，系统的可依赖性有 4 个主要方面：

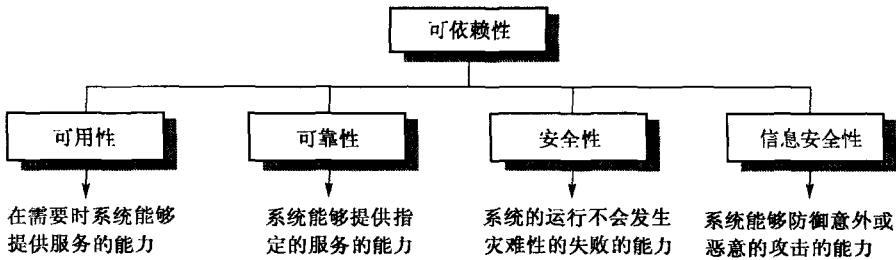


图 11-1 主要的可依赖性特征

1. 可用性 一般来讲，系统的可用性是指系统在任何时间都能运行并能够提供有用服务的可能性。

2. **可靠性** 一般来讲,系统的可靠性是系统在给定的时段内能正确提供用户希望的服务的可能性。

3. **安全性** 一般来讲,系统的安全性是判断系统将会对人和系统的环境造成伤害的可能性。

4. **信息安全性保密性** 一般来讲,系统的信息安全性是判断系统能抵抗意外的或蓄意的入侵的可能性。

图 11-1 所示的可依赖性特征都很复杂,它们可以分解为一些其他更简单的属性。例如,信息安全性包括完整性(保证系统程序和数据没有损坏)、机密性(保证信息只能由得到授权的人访问)。可靠性包括正确性(保证系统服务按照所定义的那样)、精确性(保证信息按照需要的细节层次传送)和及时性(保证信息能在规定的时间内传送到)。

当然,这些可依赖性属性不是对所有系统都是同样重要的。在第 1 章所介绍的胰岛素泵系统中,最为重要的属性是可用性(在需要的时候必须可用)、可靠性(它必须传输正确剂量的胰岛素)和安全性(决不能传输危险剂量的胰岛素)。在这里,信息安全性就不是个问题了,因为泵不维护机密信息,也不在网络环境下工作,所以不会受到恶意攻击。对于野外气象系统,可用性和可靠性是最重要的特性,因为维修的费用是非常高的。对于病人信息系统,信息安全性是特别重要的,因为其维护的都是敏感的私人数据。

除了这 4 个主要方面外,我们还可以认为可依赖性包括如下其他的一些性质:

1. **可维修性** 系统失败是不可避免的,但是如果系统可以很快修复的话,系统失败而导致的崩溃就可以尽量避免。为了做到这一点,就必须能诊断问题,找到失败的组件并加以修复。提高软件的可维修性需要使用系统的机构能够获取软件源代码,并有修改代码的能力。开源软件在这这一点上更容易一些,但是对组件的复用会使得做到这一点更困难。

2. **可维护性** 在系统使用过程中新需求会不断出现,改变它使之保持有用性,不断适应新需求就变得相当重要。可维护软件能够以低成本的修改来应对新需求,而且在修改过程中引入新错误的可能性比较小。

3. **生存能力** 基于互联网的系统的非常重要的属性就是生存能力(Ellison 等, 1999b)。生存能力是系统在受到攻击的情况下甚至在部分系统已经瘫痪的情况下能继续提供服务的能力。提高生存能力主要是识别关键系统组件并保证它们能提供最低服务。有 3 个策略用于增强系统生存能力:抵御攻击、攻击识别和从攻击造成的毁坏中恢复(Ellison 等, 1999a; Ellison 等, 2002)。第 14 章将更详细地讨论这一点。

4. **容错** 这可以看成是可用性的一部分,反映为在多大程度上系统的设计能避免用户输入错误并在输入有错时能够容忍错误的存在。当用户发生错误时,系统应该尽量检测这些错误,而且要么能够自动修改错误要么请求用户重新输入数据。

系统可依赖性的概念是作为一个全面的属性来介绍的,因为它的可用性、信息安全性、可靠性、安全性这些属性都是紧密联系的。安全的系统运行通常需要系统是可用地并且可靠地运行。一个系统可能由于一个入侵者破坏了它的数据而变得不可靠。对一个系统的拒绝服务攻击其目的是损坏系统的可用性。如果一个系统被病毒感染了,你就不能再对它的可靠性和安全性有信心,因为病毒可能会改变系统的行为。

为开发一个可靠软件,你需要做到:

1. 避免在软件描述和开发过程中引入意外的错误。
2. 设计检验和有效性验证过程,使之能够有效地发现影响系统可靠性的残余错误。
3. 设计保护机制防范能够损坏系统的可用性和信息安全性的外部攻击。
4. 正确地配置部署的系统和支持软件,以提供良好的运行环境。

另外,你通常应该假设你的软件并不完美,并且软件失败有可能会发生。因此你的系统应该

包含能够尽快恢复通用系统服务的复原机制。

容错的需求意味着可靠的系统必须包含冗余代码用以监视系统本身，探测错误状态，并且在失败发生之前从错误中恢复。这会影响到系统的表现，当每次系统执行时都要做额外检查。因此，设计者总是要在系统性能和可依赖性两者之间做出折中。你可能因为系统运行速度变慢而弃用系统检查。然而，随之而来的风险是由于一些缺陷没有被发现而导致发生一些失败。

因为额外的设计、实现和有效性验证成本，加强系统可依赖性会大大增加开发成本。尤其是对于需要过度可依赖性的系统，如安全性要求极高的控制系统，有效性验证的成本会非常高。除了需要检验系统符合其需求定义外，还需要向外部管理者证明系统是安全的。例如，航空器系统必须向外部管理者比如国家航空管理局证明系统是安全的，影响到飞行器的安全的灾难性系统失败发生的概率是极端低的。

图 11-2 给出了成本和对可依赖性渐增改善之间的关系。如果你的软件可依赖性不是很高，你可以通过更好的软件工程方法用较低的代价显著提高你的软件。然而如果你已经使用了很好的措施，改进的花费将高很多而得到的效益却比较低。还有一个问题是测试你的软件并证明你的软件是可靠的也绝非易事。这需要运行多次测试并查看一定数量发生的错误。随着你的软件可依赖性变得更高，你将会看到越来越少的失败。结果，需要更多的测试并猜测你的软件还有多少问题。测试是非常昂贵的，这急剧增加了高可用性系统的成本。

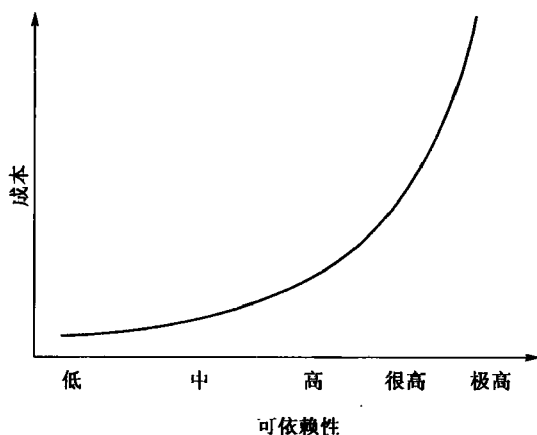


图 11-2 成本/可依赖性曲线

11.2 可用性和可靠性

系统的可用性和可靠性是两个关系紧密的可依赖性指标。这两个属性都可以用概率来表达。系统可用性是系统在请求发生时响应并提供这些服务给用户的可能性。系统的可靠性是系统能按需求描述中所定义的那样正确提供所需服务的可能性。如果，平均来说，每 1000 次输入中两次输入引起失败，那么其可靠性，用发生失败的频率来表示，就是 0.002。如果可用性是 0.999，即表示在一段时间里的 99.9% 都是可用的。

可靠性与可用性是紧密联系的但是有时候一个比另一个更重要。如果用户期望从系统得到持续的服务那么这个系统就有高可用性需求。无论何时当一个要求产生时它必须是可用的。然而，如果由于失败而造成的损失不大并且系统可以迅速恢复那么这些失败不会严重地影响系统使用者。在这样的系统当中，系统的可靠性需求可能相对较低。

交换机连接电话呼叫线路是一个系统中可用性比可靠性更重要的例子。当拿起听筒的时候，用户希望听到拨号音，因此系统有较高可用性需求。如果当一次连接被建立起以后系统发生了失败，错误可以很快得到修复。交换器通常可以重置系统并重新尝试连接。这可能发生得非常之快，以至于用户甚至没有注意到失败曾经发生。再者，即便是通话真的被打扰了，其造成的结果也通常不很严重。因此，在这一类系统中可依赖性的关键要求不是可靠性而是可用性。

系统可靠性和可用性的更精确的定义如下：

1. **可靠性** 系统在一特定时间特定环境中为一专门目的而做的无失败操作的可能性。
2. **可用性** 系统在一个时刻是可操作的和能执行请求服务的可能性。

在开发可依赖的系统时存在的一个实际的问题是，我们对可靠性和可用性的直觉概念时常

比这些有限的定义宽广得多。在可靠性的定义陈述中包含系统被使用的环境和它使用的目的两个条件。因此，一个环境中的可靠性度量就不适合另一个环境中该系统的度量。

举例来说，一个软件系统的可靠性度量是在办公室环境中进行的，环境中绝大多数用户都对软件的操作不感兴趣。他们会按照说明书一板一眼地操作，从不对系统做其他试验。在一个大学环境中，可靠性可能就大大不同了。在这里，学生探究系统的边界，可能以一种意想不到的方式使用系统。这些都会造成系统的失败，而在办公室环境中很少发生这种情况。

这些标准的可用性和可靠性的定义没有考虑到失败的严重程度和不可用性的后果。人们通常接受微小的系统失败但是很担心能够造成严重后果的严重失败。比如，损坏存储数据的计算机失败比可以通过重启机器解决问题的机器死机的失败更难以接受。

可靠性的严格定义是与系统描述的实现相关的。这就是说，如果系统的行为是与描述中定义的严格一致，则认为系统行为是可靠的。不过，通常存在的一个能感知到系统不可靠的原因是系统定义与系统用户期待不匹配。不幸的是，许多描述都是不完善的或不正确的，软件工程师根据描述来构造系统的行为，因而所得到的系统行为当然会存在问题。因为他们不是领域专家，所以他们不可能准确实现用户预期的行为。当然可以肯定的是，用户不去读系统描述，他们可能因此对系统有不现实的期望。

可用性和可靠性显然是有联系的，因为系统失败可能导致系统的损坏。然而，可用性不只取决于系统失败的次数，同时取决于修复导致失败的错误所需时间。因此，如果系统 A 每一年失败一次，系统 B 每个月失败一次，则 A 系统比 B 系统相对更可依赖。然而，如果系统 A 需要花 3 天才能重新启动，而系统 B 只需 10 分钟就可以重新启动，系统 B 全年的可用性（120 分钟宕机时间）会被认为比系统 A（4320 分钟宕机时间）更具有可用性。用户或许更喜欢系统 B。

不可用系统所造成的混乱不是简单反映在系统的可用性度量上，可用性度量描述的是不可用时间所占的百分比。系统失败发生的时间点同样很重要。如果一个系统每天在凌晨 3 点到凌晨 4 点的一个小时时间不可用，并不会影响很多用户。然而，如果同样的系统在上班时间内有 10 分钟不可用，系统的不可用性可能会造成更大的影响。

系统的可靠性和可用性问题很大程度上是由系统失败引起的。这些失败有些是具体的错误的后果或其他相关项目系统中（比如，通信系统中）的失败。不管怎样，许多失败是由于系统易出的错误行为而导致的，这是系统中存在故障所致。当讨论可靠性的时候，区别术语缺陷、错误和失败是有益的。图 11-3 定义了这些术语并用一个野外气象系统的例子解释了这些定义。

术 语	描 述
人的错误和误会	人的所有导致在系统中引入缺陷的行为。例如，在野外气象系统中，程序员会决定计算下一次传输数据的时间为当前时间加 1 小时。这里的出错情况发生在当传输数据的时间在 23:00 和午夜之间（午夜时间 00:00，对于 24 小时制）
系统缺陷	软件系统的能导致系统出错的特性。缺陷是包含加 1 小时到上一次传输数据的时刻的代码，而没有检查是否时间大于或等于 23:00
系统错误	一个错误的系统状态，能够导致系统行为完全超出系统用户预期。当缺陷代码被执行时，传输数据时刻的值设置得不正确（应该为 24:xx，而不是 00:xx）
系统失败	在某一时刻所发生的事件，系统不能提供一个用户所期待的服务。没有气象数据传出因为时间是无效的

图 11-3 可靠性术语

当一个或一系列输入引起有缺陷的代码执行，一个错误的状态产生并可能引起软件失败。图 11-4 引自 Littlewood（1990），说明了一个软件系统是输入到输出集合的映射。一个程序可以有许

多输入（为简单起见，将组合输入和序列输入看成一个单一输入），程序响应这些输入产生一个或一组输出。例如，给定一个 URL 输入，Web 浏览器产生一个相应的显示 Web 页的输出。

大多数的输入不会引起系统失败。然而，有些输入或输入组合，如图 11-4 中用带阴影的椭圆 I_e 所示，引起错误输出的产生。程序的可靠性取决于作为引发一个错误输出的输入集合的数量。如果在集合 I_e 中的输入被系统频繁使用的部分执行，那么错误也就是频繁发生了。然而，如果输入集合 I_e 被很少使用的代码执行的话，那用户可能几乎不会遇到失败。

由于每个用户以不同的方式使用系统，他们对于可靠性的观点也不一样。对某个用户来讲影响系统可靠性的缺陷而对另一个用户来讲可能就不是（如图 11-5 所示）。在图 11-5 中容易出错的输入集对应为图 11-4 中标有 I_e 的椭圆部分。用户 2 使用的输入集与这个易错输入集相交，用户 2 将会体会到系统失败的滋味。用户 1 和用户 3 从不使用来自这个易错集合中的输入，对他们来讲，这个软件总是可靠的。

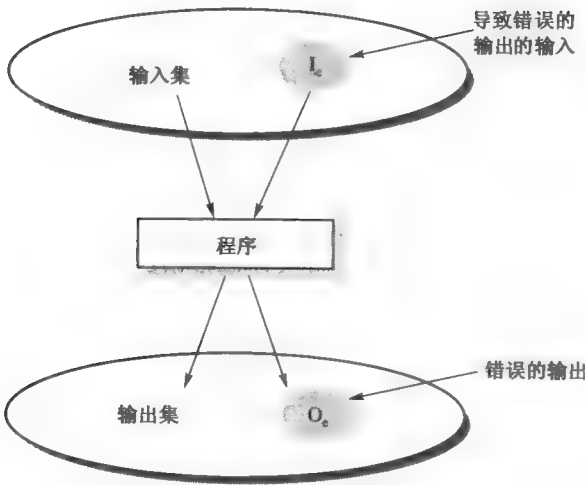


图 11-4 系统被看做是输入/输出间的映射

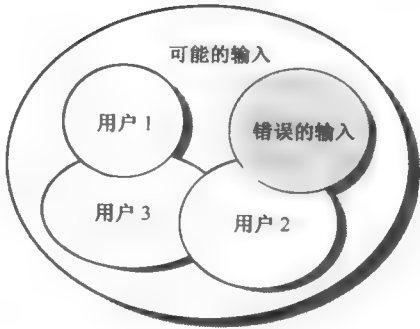


图 11-5 软件使用模式

程序真实的可靠性依赖于绝大多数用户日常使用情况下导致错误输出（失败）的输入次数。那些只发生在异常情况下的软件缺陷对系统的可靠性影响很小。结果是，对于整个系统来说，删除软件错误可能对提高整个系统的可靠性不会有显著效果。Mills 等（Mills 等，1987）发现，在他们的软件中，删除产品中 60% 的缺陷，而可靠性只提高了 3%。Adams（Adams，1984）在他对 IBM 软件产品的研究中认为：许多产品中的缺陷只在产品数百或数千个月之久的使用时段中引起一次失败。

软件的缺陷不总是导致系统错误，并且系统错误不会必然导致系统失败。其原因如下：

1. 不是程序中所有的代码都会执行。包含缺陷的代码（比如，初始化一个变量的失败）可能因软件使用的方式等原因永远不会被执行。
2. 错误是短暂的。一个状态变量可能由于缺陷代码的执行产生一个不正确的值。然而，在这个变量被访问并导致系统失败之前，其他的系统输入可能已经被处理，而此事件会重置这个状态变量为一个有效值。
3. 系统可能拥有缺陷探测和保护机制。这些机制确保在系统服务被影响之前错误行为被发现并被改正。

缺陷不一定会引起系统失败的另一个原因是，在实践中，用户会调整行为以避免使用他们知道会产生程序失败的输入。有经验的用户会依据他们的经验避开不可靠的软件特征。比如，作

者会避免特定的特征,即作者用来写这本书的字处理系统中的自动计数功能。当作者使用自动计数功能时,它经常出错。修复不使用的特征的缺陷不会改变系统可靠性。当用户共享系统问题的信息并主动绕开它们,软件中的问题的影响就会降低。

缺陷、错误和失败之间的差别在图 11-3 中给出了解释,帮助我们找出 3 个可以用于系统可靠性改善的辅助方法:

1. **缺陷避免** 使用某些开发技术尽量减少人为的错误的可能性和对人为错误采取预防手段以避免导致系统缺陷。这些技术的例子包括避免使用易出错的程序语言成分,例如指针以及用静态分析来检查程序中的异常。

2. **缺陷检测和删除** 使用检验和有效性验证技术增加发现故障的机会,并在系统被使用之前将其删除。系统性地进行测试和调试就是缺陷检测技术的一种方法。

3. **容错** 使用某些技术来确保系统缺陷不引发系统错误,同样使用这些技术确保系统错误不引发系统失败。在系统中嵌入自检设施以及使用冗余系统模块就是这样的容错技术的例子。

在 13 章中讨论了这些技术的实际应用,这其中包含了可靠的软件工程技术。

11.3 安全性

安全要求极高的系统是这样的一类系统,系统总是安全运行是至关重要的。也就是说,系统永远也不能伤害人或者损害系统环境,即使是在系统失败发生的时候。安全要求极高的系统的例子包括飞机监控系统,在化学和医学工厂中的过程控制系统,以及汽车控制系统。

安全要求极高的系统中的硬件控制相对于软件控制来说较为容易实现和分析。然而,我们现在构建此类复杂系统都不能仅仅使用硬件控制。软件控制是十分重要的,因为需要管理大量的传感器和执行器,这些单元都有复杂的控制规律。例如,先进的空气动力的失稳军用飞机要求连续的软件控制的飞行器姿态调整,以确保不会发生坠毁事件。

安全要求极高的软件会分为两大类:

1. **首要的安全要求极高的软件** 这种软件是嵌入在系统控制器中的软件。该类软件错误地执行会导致硬件的误操作,引起人员伤亡或环境破坏。胰岛素泵软件,如在第 1 章中介绍的,就是首要安全要求极高的系统。系统失败会导致用户受到伤害。

2. **次要的安全要求极高的软件** 此类软件可以间接引起人员伤亡。此类软件的例子是计算机辅助工程设计系统,它的错误执行会引起所设计的对象中存在设计缺陷。如果所设计的系统误操作的话,这样的缺陷会引起对人的伤害。另外一个例子是,心理治疗管理系统,即 MHC-PMS。由于该系统的失败,一个不稳定的病人可能没有得到正确的治疗,使得该病人伤害了自己或他人。

系统可依赖性和系统安全性是相关的,但是可依赖的系统也许是不安全的,反之亦然。软件可能一直以一种不健康的方式工作,它因而产生的系统行为将导致意外发生。有 4 个原因说明为什么可信赖软件系统不一定是安全的系统:

1. 我们不能百分百地确信软件系统是无缺陷的和能容错的。未被发现的缺陷可能潜伏很长一段时间,软件失败也许会在多年的可信赖运行之后。

2. 需求描述会是不完备的,因为它可能没有描述在一些关键时刻的必要的系统行为。高比例的系统误操作(Boehm 等 1975; Endres, 1975; Lutz, 1973; Nakajo and Kume, 1991)来自于描述错误而非设计错误。在嵌入式系统错误的研究中,Lutz 总结说:

……需求中的困难是引起安全类软件错误的主要原因,它会一直存在着直到集成和系统测试。

3. 硬件误动作会引起系统无法预测的行为,使得软件所处的环境无法预测。当组件与物理失败很紧密时,它们会产生不规则的行为,其产生的信号会超出软件所能处理的范围。

4. 系统操作员会产生单独看起来是正确的输入，但是在某些情形下它是可能导致系统误操作的。一个有趣的例子是，一架飞机在停机坪上突然起落架塌下。显然，机械师按下了指示设施管理软件收起起落架的按钮。软件完美地执行了机械师的指令。然而，系统应该不允许该指令的执行除非飞机已经升空。

形成了一组专门的词汇用来讨论安全要求极高的系统，它对于理解所使用的专业术语是很重要的。图 11-6 总结了一些重要的术语定义，同时给出了来自胰岛素泵系统的例子。

术 语	定 义
意外（事故）	未预计到的事件和事件后果，它导致人员伤亡、财产损失或环境破坏。过量注射胰岛素是意外的例子
危险	潜在能引起或造成意外发生的情况。测量血液血糖含量的传感器失败是危险的一个例子
损害	关于事故所造成的损失的一个度量。损害可以是很多人在事故中遇难，也可能是很小的身体外伤或很小的财产损失。胰岛素过量是个很严重的损害甚至引起患者的死亡
危险严重性	来自特别危险最坏可能的损害的评估。危险严重程度可以是灾难性的，很多人遇难，也可能是很小的，如只有微不足道的损失。只要有可能造成人的死亡，那危险程度就将是“非常高”
危险概率	发生危险的可能性。概率值可能是太绝对，所以可以用“可能”（有 1/100 的危险发生概率）到“不太可能”（没有可想到的发生危险的情形）等来衡量危险可能。在胰岛素泵中传感器失败导致过量注射的概率就是“较低”
风险	这是系统会造成意外的概率的度量。风险评估是考虑危险概率、危险严重性，以及危险演化为事故的概率。胰岛素过量的风险是中度到低

图 11-6 安全性术语

确保安全性的关键在于保证要么意外不会发生要么意外发生的后果并不严重。这可以通过 3 种互补的手段达到：

- 1. **危险避免** 系统的设计要能避免危险的发生。例如，切割系统要求操作员使用双手同时分别按住一个按钮以避免操作员的手在切刀的路径上。
- 2. **危险检测和去除** 系统的设计要能检测危险，并在其发生之前去除危险。例如，一个化工厂系统会检测过大的压力并开启减压阀在爆炸发生之前降低这些压力。
- 3. **限制损失** 系统会包含保护特征以最小化损失。例如，飞机引擎正常情况下带有自动的灭火器。如果失火，通常是可以其在对飞机造成威胁之前得到控制。

意外最容易发生在多个方面同时出错的时候。关于严重意外事件的分析（Perrow, 1984）表明，它们几乎都是源于系统不同部分失败的组合情形。对子系统失败的未预料的组合导致整个系统失败的交互。例如，空调系统的失败可能导致过热，而这又会导致系统硬件产生不正确的信号。Perrow 建议：不太可能预测所有可能的失败组合。意外因而是使用复杂系统不可避免的一部分。

某些人用此论断反对软件控制。因为软件的复杂性，在系统的不同部分间存在更多的交互，这意味着会有更多的缺陷组合导致系统失败。

然而，相对于机电系统，软件控制系统会监视更大范围的状态。它们也更容易进行修改。使用的计算机硬件具有非常高的内在的可靠性也在物理上减少了体积和重量。软件控制系统能提供复杂的安全互锁机制，可以支持相应策略以减少人们在危险环境中停留的时间。尽管软件控制会导入更多的错误，它也允许更好地监控和保护，因而对改善系统安全性有帮助。

在任何情况下，最重要的一点是理解系统安全的程度。系统不可能是百分百地安全的，社会

不得不决定是否某个意外突发事件会产生恶劣后果，必须要动用先进的技术去处理。同时如何部署有限的国家资源来减少民众的风险，也属于社会和政治决策。

11.4 信息安全性

信息安全性是一个反映系统保护自己免受外部意外或故意攻击的能力的一个系统属性。这些外部攻击可能原因是，大多数一般用途的电脑现在都已经联网并且因此可以被外界访问。攻击的例子通常包括：病毒或者木马的安装，系统服务的未经许可的使用，系统或数据的未经许可的修改等。如果你真的想要一个保密的系统，最好不要连接互联网。这样，你的信息安全性问题就局限于确保授权用户不会滥用系统。然而在实践当中，对大多数系统来说连接网络的利益是巨大的，所以与互联网断开连接是不划算的。

在有些系统中，信息安全性是可依赖性中最为重要的内容。军事系统、电子商务系统以及机要信息处理和交换系统都需要具有很高的保密等级。比方说，如果没有机票预订系统是很不方便的，会耽误机票出售。不过，如果系统是不安全的，一个攻击可能删除所有订票，那么日常的航班运行将在现实中无法继续。

在可依赖性的其他方面，有一个跟信息安全性有关的专门术语集。一些重要的术语，就如 Pfleeger (Pfleeger, 2007) 讨论过的，在图 11-7 中定义。图 11-8 采用图 11-7 中定义的信息安全性概念，展示了这些术语是怎样与 MHC-PMS 中的脚本相关联的。

术 语	定 义
资产	那些有价值的需要保护的东西。资产可以是软件本身也可以是系统使用的数据
暴露	对计算系统的可能产生的损失或伤害。它可以是对数据的损失或损害，或者是当信息安全失败之后进行必要恢复所需投入的时间和人力
脆弱性	基于计算机系统的一些能造成损失或伤害的薄弱环节
攻击	对系统脆弱性的利用。一般来讲，这是来自系统外部的，是故意产生某些损害的企图
威胁	具有潜在引发损失或伤害的情况。可以把它看成在攻击发生时的系统脆弱性
控制	降低系统脆弱性的保护措施。编码就是一个控制的例子，以减小弱访问控制系统的脆弱性

图 11-7 信息安全术语

术 语	实 例
资产	每个病人的记录，包括正在接受的治疗或已经接受的治疗
暴露	诊所的经济损失，未来病人不再寻求治疗，原因是他们不再相信诊所能够保护好他们的信息。来自体育明星的诉讼的经济损失。诊所失去声誉。
脆弱性	弱的密码系统使得用户非常容易猜测到密码。用户名与其真实名字一致。
攻击	假冒授权用户
威胁	一个未经授权的用户将获得访问系统的资格，他/她是通过猜测某个授权用户的用户名和密码进入系统的
控制	一个密码检查系统，不允许那些一般在词典中能找到的名字或词作为密码

图 11-8 信息安全术语的例子

诊所职员在 MHC-PMS 上用用户名和密码登录。该系统要求至少 8 个字母长的密码，但允许设置任何密码而不做进一步检查。一个嫌疑犯发现一个收入很高的体育明星正在接受心理健康

问题的治疗。他想要非法得到系统中的信息，这样他可以敲诈那位明星。

他假冒明显的亲属并与心理健康诊所的护士们谈话，他发现了如何能接触到系统和护士的个人信息。通过查看名牌他知道了能够进入系统的人员的名字。接着他用这些名字并按规律猜出可能的密码（比如孩子的名字）伪装登录系统。

对任何一个联网系统，有3个主要类型的信息安全性威胁。

1. **对系统的保密性以及它的数据的威胁** 这些威胁会把信息暴露给未经授权访问该信息的人及程序。
2. **对系统和数据的完整性的威胁** 这些威胁会毁坏或损害软件及其中的数据。
3. **对系统的有用性和系统数据的可用性的威胁** 这些威胁会限制授权用户使用软件和其中的数据。

当然，这些威胁是相互依赖的。如果一次攻击使得系统不可用，那么你将不能够实时更新信息。这意味着系统的完整性可能受到了损害。如果一次攻击成功并且完整性受到损害，那么系统必须停止使用并修复问题。因此，系统的可用性就降低了。

在现实中，大多数社会技术系统的脆弱性是由人的失败而不是技术问题造成的。人们选择容易被猜到的密码或在他们能够找到的地方写下他们的密码。系统管理员在设定访问控制或配置文件时犯了错误，用户没有安装或使用保护软件。然而，正如10.5节所讲的那样，我们在把一个问题归类为用户错误的时候必须非常小心。人的问题通常反映出系统设计决策的不足。比如，经常性改变密码（所以用户要记录下他们的密码）或复杂的设置机制。

增强系统的信息安全性控制类似于增加可靠性和安全性：

1. **脆弱性避免** 用来确保攻击不成功的控制方法。这里使用的策略是设计系统使信息安全性问题得到避免。比如说，敏感的军用系统不会连接到公用网络，这样外部访问就是不可能的。你同样应该考虑使用加密控制手段。任何未经授权的对加密文件的访问意味着其不能被攻击者阅读。在现实中，破解强化的加密文件是非常昂贵并且费时的。
2. **攻击检测和压制** 控制的意图在于检测和压制攻击。这些控制包括系统中带有某些功能，如监视其操作和检查不寻常的模式的活动。如果检查到这些，可能接下来要采取行动，比如关闭部分系统，限制某些用户的访问，等等。
3. **暴露限制与恢复** 支持从问题中恢复的控制。这些控制可以是自动备份的策略和信息“镜像”，也可以是保险措施，弥补一次成功攻击系统所造成的损失。

没有一个合理级别的信息安全性，我们就不可能对一个系统的可用性、可靠性和安全性有信心。证明可用性、可靠性和信息安全性的方法假设运行的软件与初始安装的软件是类似的。如果系统受到攻击使得软件在某种方式上被破坏（比如，如果软件被修改包含了蠕虫病毒），那么可靠性和安全性的证据就不再成立。

系统开发的错误可能导致信息安全性的漏洞。如果一个系统对意想不到的输入不能做出反应或数组范围没有检查，那么攻击者可以利用这些弱点来获取访问这个系统的机会。主要的信息安全性事故，比如最初的网络蠕虫（Spafford, 1989）和十多年以后的代码——红色蠕虫病毒（Berghel, 2001）都是利用了同一种弱点。就是利用在C#程序中不存在数组越界检查，所以有可能重写部分内存从而未经授权进入系统。

要点

- 要求极高的系统失败能造成重大的经济损失：重要信息的丢失，人身伤害，甚至危及人的生命。
- 计算机系统的可依赖性反映用户对系统信任程度的一个系统属性。可依赖性中最重要的

方面是可用性、可靠性、安全性和信息安全性。

- 系统的可用性是指系统在用户请求服务时能提供服务的可能性；可靠性是指系统能按照指定的要求提供服务的可能性。
- 理解可靠性与在正常使用中发生错误的可能性是相关的。一个程序可能包含已知的缺陷，用户可能从不使用那些受缺陷影响的系统特征。用户因而认为系统是可靠的。
- 系统的安全性是反映系统无论怎样操作都不会对人和环境带来危害的系统特性。若安全性是要求极高的系统的一个基本属性时，我们称这个系统是一个安全性要求极高的系统。
- 信息安全性反映的是系统保护自己不受外界攻击的能力。信息安全性失败可能导致可用性的损失，系统或其数据的损坏，或信息泄露给未经授权的人。
- 没有一个合理程度的信息安全性，系统的可用性、可靠性和安全性都会由于外部入侵而遭受损失。如果系统是不可靠的，确保系统安全性或信息安全性是很难得，因为它们可能因系统失败而损坏。

进一步阅读材料

《The evolution of information assurance》，这是一篇讨论机构在遭受意外事件和攻击时保护要求极高信息的需要的好文章（R. Cummings, IEEE Computer, 35 (12), December, 2002）。<http://dx.doi.org/10.1109/MC.2002.1106181>。

《Designing Safety Critical Computer Systems》，这是一篇很好的介绍安全性极高系统的领域的文章，它讨论了危害与风险的基本概念。比 Dunn 的关于安全性极高系统的书更容易理解。（W. R. Dunn, IEEE Computer, 36(11), November 2003.）<http://dx.doi.org/10.1109/MC.2003.1244533>。

《Secrets and Lies: Digital Security in a Networked World》，这是一部非常棒而且可读性很强的、关于计算机信息安全性的书，作者从社会技术焦点对此进行了深入探讨。Schneier 在论坛上发表的公众的信息安全性问题（下面是网址）同样非常好（B. Schneier, 2000, John Wiley & Sons）。<http://www.schneier.com/essays.html>。

练习

- 11.1 给出软件可依赖性对于大多数社会技术系统十分重要的 6 个理由。
- 11.2 什么是系统可依赖性中最重要的方面？
- 11.3 为什么用于保证系统可依赖性的成本随着可依赖性要求增加呈指数增长？
- 11.4 假设有如下一些系统，你认为哪个可依赖性属性对于它们是最为紧要的，说明理由。
 - 由 ISP 提供的拥有数千用户的互联网服务器。
 - 用于锁眼手术的计算机控制手术刀。
 - 用于卫星运载火箭的方向控制系统。
 - 基于互联网的个人账户管理系统。
- 11.5 指出 6 个客户产品，它们可能被安全要求极高的软件系统控制。
- 11.6 可靠性和安全性是可依赖性的既相互关联又有明显区别的两个属性。描述一下两者之间的最主要差别，并解释为什么一个可靠系统可能是不安全的，反之亦然。
- 11.7 在一个用射线来治疗肿瘤的治疗系统中，给出系统中可能存在的一种危险，提出一个软件特性用它来保证该危险不造成意外事件。
- 11.8 解释为什么在系统可用性和系统信息安全性之间存在密切关系。
- 11.9 以 MHC-PMS 为例，说明 3 种对系统的威胁（作为图 11-8 所示威胁的补充）。提出可能加入的控制手段来降低这些威胁的成功攻击的几率。
- 11.10 作为一名计算机信息安全的专家，收到某个组织的请求，需要帮助该组织未经授权地访问某个美国

公司的计算机系统。这将帮助他们证实或否定该公司正在销售此设备直接用于对政治犯的刑讯逼供。讨论该请求所引发的道德困境和应该如何回应此请求。

参考书目

- Adams, E. N. (1984). 'Optimizing preventative service of software products'. *IBM J. Res & Dev.*, **28** (1), 2-14.
- Berghel, H. (2001). 'The Code Red Worm'. *Comm. ACM*, **44** (12), 15-19.
- Boehm, B. W., McClean, R. L. and Urfig, D. B. (1975). 'Some experience with automated aids to the design of large-scale reliable software'. *IEEE Trans. on Software Engineering.*, **SE-1** (1), 125-33.
- Ellison, R., Linger, R., Lipson, H., Mead, N. and Moore, A. (2002). 'Foundations of Survivable Systems Engineering'. *Crosstalk: The Journal of Defense Software Engineering*, **12**, 10-15.
- Ellison, R. J., Fisher, D. A., Linger, R. C., Lipson, H. F., Longstaff, T. A. and Mead, N. R. (1999a). 'Survivability: Protecting Your Critical Systems'. *IEEE Internet Computing*, **3** (6), 55-63.
- Ellison, R. J., Linger, R. C., Longstaff, T. and Mead, N. R. (1999b). 'Survivable Network System Analysis: A Case Study'. *IEEE Software*, **16** (4), 70-7.
- Endres, A. (1975). 'An analysis of errors and their causes in system programs'. *IEEE Trans. on Software Engineering.*, **SE-1** (2), 140-9.
- Laprie, J.-C. (1995). 'Dependable Computing: Concepts, Limits, Challenges'. FTCS- 25: 25th IEEE Symposium on Fault-Tolerant Computing, Pasadena, Calif.: IEEE Press.
- Littlewood, B. (1990). 'Software Reliability Growth Models'. In *Software Reliability Handbook*. Rook, P. (ed.). Amsterdam: Elsevier. 401-412.
- Lutz, R. R. (1993). 'Analysing Software Requirements Errors in Safety-Critical Embedded Systems'. RE'93, San Diego, Calif: IEEE.
- Mills, H. D., Dyer, M. and Linger, R. (1987). 'Cleanroom Software Engineering'. *IEEE Software*, **4** (5), 19-25.
- Nakajo, T. and Kume, H. (1991). 'A Case History Analysis of Software Error-Cause Relationships'. *IEEE Trans. on Software Eng.*, **18** (8), 830-8.
- Perrow, C. (1984). *Normal Accidents: Living with High-Risk Technology*. New York: Basic Books.
- Pfleeger, C. P. and Pfleeger, S. L. (2007). *Security in Computing, 4th edition*. Boston: Addison-Wesley.
- Spafford, E. (1989). 'The Internet Worm: Crisis and Aftermath'. *Comm. ACM*, **32** (6), 678-87.

可依赖性与信息安全性描述

目标

本章的目标是要说明如何定义功能性和非功能性的可依赖性和信息安全性需求。读完本章，你将了解以下内容：

- 理解一个风险驱动的方法是如何识别和分析安全性、可靠性和信息安全性需求的；
- 理解缺陷树是如何帮助分析风险和导出安全性需求的；
- 引入了对可靠性描述的量度以及是如何用这些量度来描述可靠性需求；
- 知道不同种类的可能在复杂系统中需要的信息安全性需求；
- 了解对系统使用形式化的、数学的描述的优点和缺点。

1993 年 9 月，一架飞机在风暴中降临波兰华沙机场。开始降落后的 9 分钟里，飞机计算机控制的制动系统突然失灵。制动系统并未检测出飞机已经着陆了并认为飞机还在空中。飞机的安全特性中止了逆推力系统的工作，因为逆推力系统可以降低飞机的速度，它在空中启动将是危险的。飞机脱离跑道向外冲去，撞上了泥土堤岸，引起大火。

对于事故的调查显示制动系统软件按其描述正常工作了。程序中是没有错误的。然而，软件的描述是不完备的，它没有考虑到极特殊的情况，恰巧这种极特殊情况发生了。软件正常工作但是系统失败了。

这说明了系统的可依赖性不仅依赖于好的工程过程，它还需要关注一些细节，比如系统需求的导出，以及所包含的用来保证系统的可依赖性和信息安全性的特殊的软件需求。那些可依赖性和信息安全性需求分为两种：

1. **功能性需求** 定义了系统应该包含的检查和修复措施，以及防止系统失败和外部攻击的保护性特征。

2. **非功能性需求** 定义了系统需要的可靠性和可用性。

产生功能性的可依赖性和信息安全性需求的起始点通常是商业和领域内规则、政策或法规。它们是高级别的需求，最好用“不应该”来描述需求。与定义系统应该进行什么的功能性需求相比，“不应该”需求定义系统不能被接受的行为。“不应该”需求的例子是：

- 系统不应该允许用户对不是由他们建立的任何文件进行存取权限的修改（信息安全性）；
- 当飞机在飞行时，系统不应允许选择反向推力模式（安全性）；
- 系统不应该允许 3 个以上报警信号同时被激活（安全性）。

这些“不应该”需求不能直接实现，而是必须分解成更多的比较专门的软件功能需求。它们可能通过系统设计决策来实现，这样的设计决策的例子是：决定在一个系统中用特别类型的设备。

12.1 风险驱动的需求描述

可依赖性和信息安全性需求可以认为是保护性需求。它们描述了一个系统怎样保护自己不受内部缺陷破坏，阻止系统失败对其运行环境造成损害，阻止来自系统运行环境的事故或攻击破坏系统，以及在失败事件发生时的功能恢复。为发现这些保护性需求，你需要理解对于系统及

其运行环境的风险。一个风险驱动方法的需求描述应该考虑到：可能发生的危险事件，这些事件可能真正发生的概率，这样的一个事件可能引起的损失，以及损失可能波及的范围。然后根据危险事件可能造成损失的分析结果，可以建立信息安全性和可依赖性需求。

风险驱动描述是安全性和信息安全性要求极高的系统的开发者广泛使用的方式。它注重那些可能造成最大破坏或可能经常发生的事件。仅造成不严重的后果或者极少发生的事件可能被忽略。在安全性要求极高的系统中，风险与引发事故危险相关联；在信息安全性要求极高的系统中，风险来自对系统脆弱性的内部或外部的攻击。

一般性的风险驱动的描述过程（见图 12-1）包括：理解系统所面对的风险、发现它们的根源、生成需求来管理这些风险。这个过程步骤是：

- 1. **风险识别** 识别出对系统的潜在的风险，这依赖于系统使用的环境。风险可能是由系统与其操作环境的特殊情况的交互引起的。前面讨论过的华沙的飞机事故发生在雷暴天气中，侧风导致飞机倾斜，所以（不常见地）它以一个轮子着陆而不是两个轮子着陆。
- 2. **风险分析和分类** 每一个风险被分门别类地考虑。将那些具有潜在严重性和明显的危险挑选出来以便进一步分析。在这个阶段，将那些不太可能发生的危险（例如，闪电和地震同时发生）或者不会被软件探测到的风险排除掉。
- 3. **风险分解** 对每个风险单独分析以找出风险根源。风险根源是系统失败的原因。它们可能是软件或硬件错误或由系统设计决策导致的内在的脆弱性。
- 4. **风险降低** 对发现的风险提出相应的降低或根除的办法。这些将生成系统的可依赖性需求，即明确定义了对风险的防范，以及当风险产生时如何管理风险。

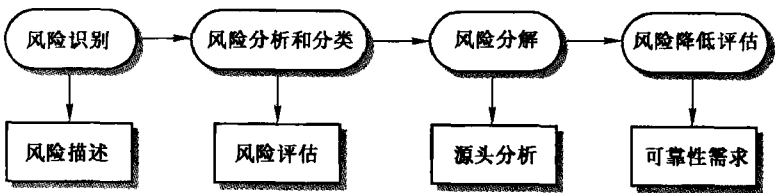


图 12-1 风险驱动的描述

对于大型系统来说，风险分析通常被分解为若干个阶段（Leveson, 1995），每个阶段强调不同类型的风险：

- 1. 初步危险分析，本阶段系统环境的主要风险将被识别出来。这些不依赖于系统开发中用到的技术。初步危险分析的目标是开发一个系统的信息安全性和可依赖性需求的初始集。
- 2. 生命周期风险分析，在系统开发期间进行风险分析，并关注源于系统设计决策的风险。不同的技术和系统体系结构有它们自己的相关的风险。在这个阶段，你应该扩展需求来预防这些风险。
- 3. 运行态风险分析，考虑系统用户界面因素以及操作人员操作错误的风险分析。而且，一旦用户界面设计决策做出，进一步的保护需求必须被加入。

这些阶段是必要的，因为在没有完整的系统实现信息基础上做出所有的可依赖性和信息安全性决定是不可能的。信息安全性和可依赖性需求特别受技术选择和设计决策影响。必须引入系统检查来保证第三方组件是正确运行的。必须修改信息安全性需求，因为它们与一个现有系统提供的信息安全性特性冲突。

比如，一个信息安全性需求可能是用户应该用一个密码短语确认身份而不是用一个密码。密码短语被认为比密码更加有信息安全性。它们对于攻击者来说更难猜，也很难用自动密码破解系统来破译。然而，如果一个决定要求用一个只允许密码验证的系统，那么这个信息安全性需

求就不能被支持。这个系统可能因此需要包含一个附加功能来补偿用密码而不用密码短语所增加的风险。



IEC 的安全性管理的标准

IEC (国际电工委员会) 为保护系统 (比如在一些危险情况发生时系统能激活自我保护功能) 定义了一个安全性管理的标准。保护系统的一个例子是当一列火车穿过红色信号灯时能自动停下来。该标准包括了大量有关安全性描述过程的指导。

<http://www.SoftwareEngineering-9.com/Web/SafetyLifeCycle/>

12.2 安全性描述

安全性要求极高系统是指当系统失败时可能影响系统环境并导致在这个环境下人员伤亡的系统。安全性描述主要关注的是识别出能将系统失败发生概率降到最小的需求。安全性描述首要的是保护性需求, 与普通的系统操作没有关系。它们可能明确说明系统应该被关闭, 这样安全性才能得以保持。在导出不同的安全性需求中, 你需要在安全性和功能性之间找到一个可接受的平衡来避免过度保护。在一个花费合理的前提下研究构建非常安全的系统才是有意义的。

回忆第10章所讨论的专业术语, 危险是可能 (但不是必须) 造成人员伤亡, 而风险是系统可能会进入一种危险状态。因此安全性描述通常是关注在一定条件下可能产生的危险, 以及可能导致这些危险产生的事件。

一般的基于风险的描述过程的活动 (如图12-1所示), 会映射到下列安全性描述过程:

1. **风险识别** 在安全性描述中, 这是识别可能危害系统的危险的过程。
2. **风险分析** 这是一个危险评估过程, 以决定哪一个危险是最危险并且 (或) 最可能发生的。应该在给安全性需求分类时对其进行优先级排序。
3. **风险分解** 这个过程是关于发现能导致危险发生的事件。在安全性描述中, 这个过程被认为是风险分析。
4. **风险降低** 这个过程基于风险分析的结果并识别安全性需求。这些可能是关于确保一个危险不会引起或导致事故, 或如果一个事故发生了将相关的损坏最小化。

12.2.1 危险识别

在安全要求极高的系统中, 主要的风险来自可能导致事故发生的危险。你通过考虑各种不同的种类的危险以解决危险识别的问题, 如物理危险、电子危险、生物危险、辐射危险、服务失败危险等。接下来分析每一类危险以发现可能发生的特定的危险。同时还必须识别由于不同的危险可能的混合而产生的潜在的危险。

前面章节中使用的胰岛素泵系统的例子就是安全性要求极高的系统, 因为失败可能导致患者受伤或甚至死亡。当使用机器时可能发生的事故包括: 用户承受长时间的低血糖控制的后果 (眼睛、心脏和肾脏问题); 由低血糖引起的感知功能障碍; 或者一些其他医疗情况, 比如过敏反应。

胰岛素泵系统引起的一些危险是:

1. 胰岛素药量过大 (服务失败危险)。
2. 胰岛素药量不够 (服务失败危险)。
3. 电池用尽造成停电 (电器危险)。

4. 机器与其他医疗设备发生干扰，如与心脏起搏器相互影响（电器危险）。
5. 由于不正确的安装造成传感器和执行机构的接触不良（物理危险）。
6. 机器的某个部分在病人身体内脱离（物理危险）。
7. 由于引入机器造成感染（生物危险）。
8. 患者对机器材料或胰岛素的过敏反应（生物危险）

有经验的工程师、领域专家以及专业的安全顾问一起工作，识别出系统风险。集体工作方式比如头脑风暴，在识别风险中会有很好的作用。对先前事故有直接经验的专门分析人员也能够识别特定的风险。

软件相关的风险一般关心的是系统服务失败、或者是监视系统或保护系统失败。监视系统会检测出潜在的危险状态，比如断电。

12.2.2 危险评估

危险评估过程注重理解危险发生的频率和危险发生所造成的后果。你需要进行分析来理解一个危险是不是对系统或环境的严重威胁。分析同样还要提供基本信息以决定如何管理与危险相关的风险。

对于危险，分析和分类过程的结果是可接受性报告。报告是用风险术语表述的。风险包含意外发生的可能性和它的后果。在危险评估中可以分为3种风险类型：

1. **不可容忍的风险** 在安全要求极高的系统中是指那些能威胁到人的生命的危险。系统设计不能让这类危险发生，或者一旦发生，系统特征将保证在引起事故前它们是可探知的。在胰岛素泵的例子中，一个不可容忍的风险是一次胰岛素过量注射。

2. **低于合理的实际水平（ALARP）的风险** 此类风险是那些没有太严重后果或后果严重但发生的可能性非常低的风险。系统设计应该让危险引起的事故尽量少，并服从于成本或交付等因素的约束。一个胰岛素泵的低于合理的实际水平的风险可能是硬件监控系统的失败。其后果是，在最坏的情况下，短期的胰岛素量过低。这是一个不会引起严重后果的情况。

3. **可接受的风险** 此类风险是那些相关联的事故通常引起很小的损失。系统设计者应该采取所有可能的步骤来尽量降低“可接受的”风险，只要不会提高成本，延长交付时间，或影响其他非功能性的系统属性。在胰岛素泵的例子中一个可接受风险可能是引起用户一次过敏反应。这通常仅仅引起很小的皮肤发炎。它不值得为了降低这个风险使用特别的、更昂贵的材料与设备。

图12-2（Brazendale 和 Bell，1994）给出了这两个区域。这个图的形状反映了成本与避免危险引发的事故之间的关系。应付危险的系统设计成本是这个三角形的宽度的函数。最高的成本对应到图的顶部的危险，最低的成本对应为三角形顶点处的危险。

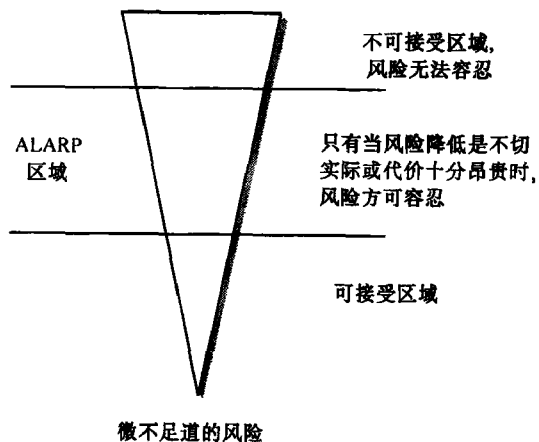


图 12-2 风险三角形

图12-2中的区域间边界不是技术上的而是取决于社会和政治因素。随着时间的推移，社会变得更加能够规避风险，所以这些边界会向下移动。虽然处理危险导致的事故的财政成本有时远小于预防这类事故所耗费的成本，但是公众可能会要求把钱花在降低系统可能发生的故事几率上，这样导致额外开销。

举例来说, 对于一个公司, 对偶然发生的污染事故的处理成本要比安装防止污染的装置的成本小。然而, 因为公众和媒体不会容忍这样的事故, 不预防而选择清理污染就变得不再能够接受。这样的事件可能还要导致风险的重新分类。比如, 认为不太可能发生的风险 (因此在 ALARP 区域中) 可能因事件发生而重新划分为不可容忍, 比如恐怖袭击, 或已经发生的事故。

危险评估过程包括估计风险可能性和风险的严重性。由于危险和事故不常见, 所以危险评估过程通常是困难的, 所以有很多工程师中没有经历过危险事件或事故的直接经验。可能性和严重性使用一些比较的语言给出, 例如“很可能”、“不太可能”、“罕见的”和“高”、“中”“低”等。如果可获得足够的危险事件和事故来作为统计分析数据, 就可能量化这些术语。

图 12-3 给出一个风险分类, 对应前面章节中讨论的胰岛素注射系统中找出的危险。把由于计算错误导致的胰岛素过量与胰岛素量过低的危险区分开来。一次胰岛素过量在短期内潜在地存在比胰岛素量过低更高的危险。胰岛素过量可以导致感知功能性障碍、昏迷、甚至死亡。胰岛素量过低引起血糖过高。在短期来看这会导致疲劳但是不会非常严重; 然而长期来看, 它们可能导致严重的心脏、肾脏以及眼睛问题。

识别的危险	危险概率	事故严重性	估计的风险	可接受性
1. 胰岛素过量计算	中度	高	高	不可接受
2. 胰岛素低于常值计算	中度	低	低	可接受
3. 硬件监控系统失败	中度	中度	低	ALARP
4. 电源失败	高	低	低	可接受
5. 机器未正常安装	高	高	高	不可接受
6. 机器失灵	低	高	中度	ALARP
7. 机器引起感染	中度	中度	中度	ALARP
8. 电子干扰	低	高	中度	ALARP
9. 过敏反应	低	低	低	可接受

图 12-3 胰岛素泵系统的风险分类

图 12-3 中的危险 4 ~ 危险 9 跟软件无关, 但是软件仍然要承担危险探测的角色。监控硬件的软件应该监控系统状态并警告潜在的问题。通常, 警告在危险导致事故之前被检测出来。可以被检测出来的危险的例子有: 电池断电和血糖传感器放置不正确。

当然, 这个监控软件是与安全相关的, 因为危险检测失败会导致事故发生。如果监视系统失败但是硬件正常工作, 那么这就不是一个严重失败。然而如果监视系统失败以至于硬件失败没有被检测到, 那么这就是更加严重的后果。

12.2.3 危险分析

危险分析是指发现安全性要求极高系统中危险根源的过程。你的目标是找出什么事件或事件的组合是引起一个系统失败的危险。为做到这一点, 你可以使用从上到下或者从下向上的研究方法。推理技术, 可能是较容易的一种方法, 这个方法是从危险开始处分析可能引发的失败; 归纳法是从一些系统失败处开始, 找出可能引起该失败的危险。只要可行, 两种方法都可以用于危险分析。

人们提出了各种各样的危险分解与分析技术。Storey (1996) 对这些做了总结, 包括审查和核对表技术、形式化分析技术, 如 Petri 网分析 (Peterson, 198); 形式逻辑 (Jahania 和 Mok, 1986) 和缺陷树分析 (Leveson 和 Stolzy, 1987; Storey, 1996) 等。这里没有足够的篇幅来讲述

所有的技术，所以集中讲解一个广泛使用的基于缺陷树的危险分析方法。这个技术很易于理解，并不需要专业领域知识。

要做缺陷树分析，先要从所有已经识别出的危险开始。对于每一个危险，要通过回溯发现可能引起该危险的原因。将危险置于树的根节点，并识别出所有能导致此危险的那些系统状态。对每一个这样的状态，接下来就是找出更多能导致该状态的那些状态。继续分解直到达到风险的所有原因。较之那些只有一个根本原因的危险，由一组根本原因的组合引起的危险导致事故的发生的可能性要小得多。

图 12-4 是关于胰岛素泵系统中的软件相关危险的一棵缺陷树。这些危险能导致不正确的胰岛素传输剂量。在这种情况下，把胰岛素不足与胰岛素过量合并为一种危险，即“不正确的胰岛素剂量使用”。这减少了所需要的缺陷树的数量。当然，在指定软件应该如何对这些危险做出反应的时候，必须区分胰岛素低剂量与胰岛素过量的区别。如前面所讲，它们不是同等重要——短期来看，过量有更大的危险。

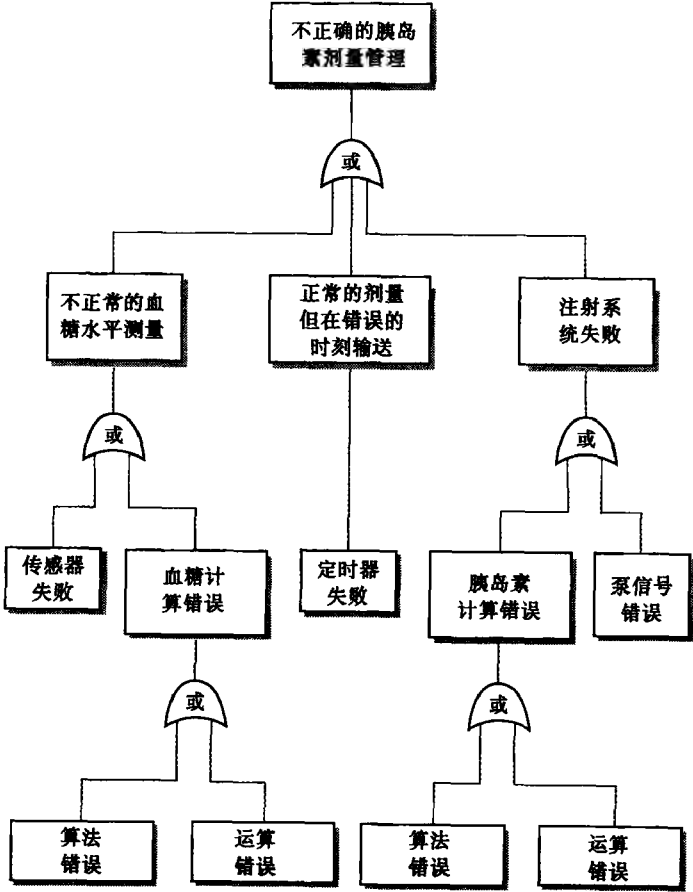


图 12-4 缺陷树的例子

从图 12-4 可以看到：

1. 有 3 种情况可以导致不正确的胰岛素剂量使用。血糖浓度可能被错误测量，所以胰岛素需求量也因错误的输入而被错误计算。传输系统可能对指定胰岛素注射量的指令没有正确反应。另外，剂量可能计算正确但是被过早或过迟传输。
2. 缺陷树的左枝是血糖值水平的错误测量。这可能是因为提供一个输入以计算血糖水平的

传感器失败，也可能是因为对血糖水平的计算执行错误。血糖值的计算是根据多个测量得到的参数，比如皮肤的传导性。不正确的计算可能是由错误的算法或者一个算术错误引起的，此错误可能因用浮点数引起的。

3. 该树的中间分支是关于时序问题的，断定这些问题只能由系统时序失败引起。

4. 树的右枝是关于传输系统失败的，检查这个失败的可能的原因。这些原因可能来自对胰岛素需求的不正确的计算，或者是因为没有能够传给胰岛素泵正确的信号。同样，一次不正确的计算可能源于算法失败或算术错误。

缺陷树也用在对潜在硬件问题的识别上。硬件错误树还可帮助理解软件需求，以检测和修正问题。例如，胰岛素的提供频率不会是很高的，一般不超过每小时三次，有时会更少。因此处理器有能力运行诊断和自检程序。在对患者产生严重后果之前，硬件错误，比如传感器、泵或者是时序错误就可以被发现并报警。

12.2.4 风险降低

一旦识别出潜在的风险及其根源，我们就能够导出安全性需求以管理风险，并确保不会发生各种事故。我们可以使用3个可行的策略：

1. **危险避免** 系统设计使得危险不能发生。

2. **危险检测和排除** 系统设计使得可以在危险导致事故之前检测出危险并排除它。

3. **灾害限制** 系统设计使得事故后果影响达到最小。

通常，安全性要求极高的系统的设计者会综合使用这些方法。举例来说，处理无法忍受的危险时，要最大可能减少其发生的可能性，并增加一个保护系统以防这个危险发生。例如，在医药工厂控制系统中，系统将努力去检测和避免反应器的压力过高。尽管如此，也还会有一个独立的保护系统来提供对压力的监视，并当检测到压力过高的时候自动开启减压阀。

在胰岛素注射系统中，“安全状态”是没有胰岛素注射的关闭状态。在短时间内这不会对病人健康带来威胁。考虑到可能导致的错误的胰岛素泵系统软件失败，下面这些内容就需要改进了：

1. **算术错误** 该错误发生在一次错误的算术计算引起的一个表示失败，在系统描述中，一定要识别出所有可能的算术错误，并为每个可能发生的错误设置异常处理程序。系统描述还要给出每当这种错误发生时所应该采取的行动。默认的安全的行动会停止注射系统并启动一个警告。

2. **算法错误** 这是一种更困难的情形，因为没有明显的必须处理的程序异常。比较要求的胰岛素剂量和之前传输的剂量这类错误能被检测到。如果高出很多，这可能表明计算出的剂量是错误的。系统也可以继续跟踪剂量序列，在有多个高于平均值的剂量被注射时，就发一个警告并限制进一步的药量注射。

针对胰岛素泵系统，我们所得到的部分安全需求列于图12-5中。它们是用户需求，自然地，它们应该更详细地表述在最后的系统描述中。在图12-5中，所提及的表3和表4都存在于需求文档中，此处并不列出。

SR1：系统所传输的胰岛素单个剂量不应该大于一个指定的最大值
SR2：系统所传输的胰岛素每日总剂量不应该大于一个指定的最大值
SR3：系统应该包括一个硬件诊断工具，该工具在每小时内应执行不少于4次
SR4：系统应该包含异常处理程序，异常处理程序处理系统中的所有识别出的异常，参见表3
SR5：当任何硬件或软件异常发生时，报警器就会发出响声，同时诊断消息（如在表4中所列）也会显示出来
SR6：在报警事件中，胰岛素传输应暂停，直到用户重新设定系统并清除报警

图12-5 安全性需求的例子

12.3 可靠性描述

正如第 10 章所述，系统总的可靠性取决于硬件可靠性、软件可靠性，以及操作员的可靠性。系统软件必须考虑到这一点。同时要包括补偿软件失败的需求，可能还要有相关的可靠性需求，以帮助解决检测和恢复硬件失败和操作员错误。

可靠性不同于安全性和信息安全性，因为它是可度量的系统属性。这就是说，指定所要求的可靠性等级，实时监控系统的运行，以及检查所要求的可靠性是否达到要求，这些是完全可能做到的。比如说，一个可靠性需求可能是系统失败要求系统重启的次数不能超过每星期一次。每次这样的失败发生都会被记录下来，这样你就可以查看可靠性需求等级是否已经达到。如果没有，你可以修改可靠性需求，也可以提交一个修改要求以应对潜在的系统问题。你可能决定接受一个较低等级的可靠性，因为改变系统以提高可靠性或者修复问题可能有负面效果，比如较低的性能或吞吐量。

比较而言，安全性和信息安全性是避免不希望的情形，而不是指定一个希望的安全性和信息安全性的“等级”。在系统的生命周期中甚至一次这样的情况都是不可接受的，而且，如果这样的情况发生了，必须做出系统变更。像“系统缺陷应该造成少于每年 10 次伤亡”的表述是没有道理的。一旦伤害发生，系统问题必须得到纠正。

因此，可靠性需求分为两种：

1. 非功能性需求，定义了在日常使用中可接受的失败的次数，或系统不可用的时间。这些是定量的可靠性需求。
2. 功能性需求，定义了系统和避免、缺陷检测或容错等软件功能，因而保证这些缺陷不会引起系统失败。

量化的可靠性需求引出相关的功能性系统需求。为达到某种可靠性需求等级，系统的功能性和设计需求应该描述缺陷并采取行动，以保证这些缺陷不会导致系统失败。

可靠性描述的过程可以基于如图 12-1 中的主要的风险驱动描述过程：

1. 风险确认 在这个阶段，要确认可能导致某种经济损失的系统失败的类型。比如说，一个电子商务系统可能因不可用而导致客户不能发出命令，或一个失败损坏了数据，需要时间来从备份中恢复系统数据库，并重新进行已经进入交易步骤的交易。在图 12-6 中显示的可能的失败类型的列表，可以作为一个风险识别的起始点。

2. 风险分析 包括了估计不同种类软件失败所造成的损失，并选择较高损失的失败进行继续分析。

失 败 类 型	描 述
无法提供服务	系统不可用，不能够传送服务给用户。你可以对它进一步分类为无法提供紧要的服务和无法提供非紧要的服务。非紧要服务失败的损失要小于紧要服务失败的损失
提供不正确的服务	系统不能正确地提供服务给用户。同样，我们可以用紧要的服务和非紧要的服务、或者小错误和主要错误来定义它们
系统/数据崩溃	系统的失败导致对系统本身或其数据的损害。这常常但不一定与其他类型失败同时发生

图 12-6 系统失败的类型

3. 风险分解 在这个阶段，要对严重的和可能的系统失败进行根本原因分析。然而，在需求阶段这可能是做不到的，因为根本原因可能依赖于系统设计决策。你可能在设计和开发阶段还要回头来做这个工作。

4. **风险降低** 在这个阶段,你应该做出定量的可靠性描述来阐明不同种类型的失败的可接受概率。这些当然都应该把失败造成的损失考虑在内。你应该为不同的系统服务使用不同的概率。你可能还要生成功能可靠性需求。同样的,这个过程可能也需要等待系统设计决策的完成。然而,正如 12.3.2 节讨论的那样,有时候做出定量的描述是很难的。你可能只能做到对功能可靠性需求的识别。

12.3.1 可靠性度量

总的来说,可靠性可以描述为系统在一个指定的操作环境中运行时发生失败的概率。如果你愿意接受,比如说,在 1000 次业务处理中有一次是失败的,那么你可以描述失败的概率为 0.001。这当然并不意味着,在 1000 次事务处理中你一定会看到一次失败。它意味着如果你观察 N 千次事务处理,你看到发生失败的次数应该是在 N 这个数附近。你可以为不同种类的失败和系统的不同部分细化这个数字。你可以决定令系统的要求很高的部分相比要求不那么高的部分有较低的失败发生概率。

用两个很重要的度量来描述可靠性,还有两个额外度量是用来描述相关系统的可用性属性。度量的选择取决于被描述系统的类别和应用领域的需求。这些度量是:

- 1. **请求失败的概率 (POFOD)** 如果使用这个度量,你就定义了系统服务请求将导致失败的可能性。所以, $POFOD = 0.001$ 意味着当做出一个请求的时候可能发生失败的概率是 $1/1000$ 。
- 2. **失败发生率 (ROCOF)** 这个度量阐明在一段时间 (比如一个小时),或在一定的系统执行的次数之内,能够观察到的系统失败的次数。在上面的例子当中, $ROCOF$ 是 $1/1000$ 。 $ROCOF$ 的倒数是失败发生的平均时间 (MTTF),其常用在可依赖性度量当中。 $MTTF$ 是观察到的系统失败时间间隔的平均值。因此, $ROCOF$ 为每小时两次暗示着平均失败时间是 30 分钟。
- 3. **可用性 (AVAIL)** 系统的可用性反映出当需要它的时候它提供服务的能力。 $AVAIL$ 是当向某一服务提出要求的时候系统的可操作性。因此, 0.9999 的可用性是指,平均来说系统在操作时间的 99.99% 是可用的。图 12-7 展示了实际中不同的可用性等级意味着什么。

可用性	解 释
0.9	系统在 90% 的时间里是可用的。这意味着,在 24 小时 (1440 分钟) 时间里,系统有 144 分钟将是不可用的
0.99	系统在 24 小时的时间里不可用的时间是 14.4 分钟
0.999	系统在 24 小时的时间里不可用的时间 84 秒
0.9999	系统在 24 小时的时间里不可用的时间 8.4 秒。大约是每周一分钟时间

图 12-7 可用性描述

当请求失败会产生一个严重的系统失败的时候, $POFOD$ 应该用做可靠性度量。这一点与提出请求的频率无关。例如,一个保护系统监控化学反应并在过加热的情况下停止反应进行,如果过加热,就应该用 $POFOD$ 定义的可靠性描述。总的来说,对保护系统的要求是不经常的,因为这一系统是在所有的修复措施失败之后的最后防线。因此 $POFOD$ 为 0.001 (1000 次请求时有一次是失败的) 看起来可能是有风险的,但如果在其生命周期中只有两三次对系统的请求,那么你可能永远见不到系统失败。

$ROCOF$ 是在对系统做有规律的请求而不是断断续续的请求时最恰当的度量。例如,在一个处理大量事务的系统当中,你可能要指定一个每天 $ROCOF$ 为 10 的失败发生率。这意味着你愿意接受平均每天 10 次事务处理不能够成功完成并必须取消的事实。要么,你可以定义失败发生率

为每 1000 次事务中失败的次数。

如果失败之间的绝对时间很重要的话，你可以定义所需要的可靠性为失败平均间隔时间。例如，如果你在为一个有很长的事务处理序列的系统定义可靠性需求（比如说计算机辅助设计系统），你应该把可靠性定义为长的失败平均间隔时间。MTTF 应该比一个用户在他的模型上连续工作而未存盘的平均时间要长得多。这将意味着用户无论在哪个工作阶段中都不太可能由于失败丢失数据。

为评估系统可靠性，你需要获取系统运行的数据。数据需求可能包括：

1. 对确定数量的系统服务请求统计系统失败的次数，这是用来测量 POFOD 指标的。
2. 系统失败平均间隔时间（或所事务处理的数目），以及总的时间或者是总的事务执行次数。这是用来测量 ROCOF 和 MTTF 的。
3. 系统失败导致不能提供服务之后的维修和重启所需的时间。这是用来测量可用性的。可用性不仅取决于失败间隔时间，而且取决于系统恢复运行的时间。

在这些度量中使用的时间是日历时间或处理机时间或其他离散单位，如事务处理的数目。在需要花许多时间来等待服务请求的响应的系统中，如电话交换系统，应该使用的时间单位是处理机时间。如果使用日历时间，那么这将包括系统什么也不做的时间。

对于连续运行的系统，你应该使用日历时间。监视系统，比如报警系统和其他种类的过程控制系统也归为这个类别。提供事务处理的系统（比如银行 ATM 机或航空预定系统）在一天的不同时候有变化的负载。在这些情况下，“时间”的单位应该是事务的数量（比如说，请求失败的可能性应该表示为每 N 千次事务中失败事务的次数。）

12.3.2 非功能性的可靠性需求

非功能性的可靠性需求是对系统可靠性和可用性的要求做出的量化描述，使用前面讲述的度量之一来计算。量化的可靠性和可用性描述多年来被用于安全性极高的系统中，但是很少用在业务要求极高的系统中。然而，随着更多的计算机向其系统要求 24/7 的服务，很可能这个技术会被更多地使用。

导出量化可靠性描述有以下几个优点：

1. 决定可靠性需要的等级的过程有助于弄清楚信息持有者到底需要什么。也帮助信息持有者理解不同种类的系统失败，也让他们清楚要达到高级别的可靠性是需要昂贵的花费的。
2. 评估何时可以停止测试一个系统。你应当在系统达到了要求的可靠性等级的时候停止测试。
3. 它是评估系统可靠性的不同设计策略的方法。你可以对每个策略如何影响可靠性需求等级做出判断。
4. 如果一个外部管理者需要在进入一个服务之前许可系统（比如，在飞行器上对飞行安全要求严格的系统），那么能说明已达到一个要求的可靠性目标的证据在系统认证中很重要。

为建立需要的系统可靠性等级，你必须考虑到系统失败可能造成的相关的损失。不仅仅是经济上的损失，还有商业上的声望的损失。声望上的损失意味着消费者会投向其他地方。虽然短期来看系统失败的损失可能比较小，但是长期的损失可能更加显著。比如说，如果你尝试访问一个电子商务网站但发现是不可用的，你可能会试着到别的地方找你想要的东西而不是一直等着这个网站直到它可用为止。如果这种情况发生了几次，你可能再也不在这个网站买东西了。

使用度量（比如 POFOD ROCOF AVIAL）描述可靠性的问题是，可能会产生过描述而因此导致高开发和高验证花费。其原因是系统的拥有者发现很难将自己的经验解释成量化的描述。他们可能认为 POFOD 是 0.001（1000 次要求中有一次失败）表示一个相对不可靠的系统。然而正

如已经讨论过的，如果对服务的要求很不常见，它很可能表示的是一个很高的可靠性。

如果你用度量来描述可靠性，评估需要的可靠性级别已经达到显然很重要。进行这种评估是系统测试的一部分。为了得到系统可靠性的数据，你需要观察一定数量的系统失败。如果你有 0.0001 的请求失败的可能性（1 万次要求中有一次失败），那么你就要设计 5 万或 6 万次请求并观察到几次失败。现实中设计并实验这么多次的实验或许是不可能的。因此，对可靠性的过描述会导致很高的测试费用。

当描述系统的可用性的时候，你可能遇到同样的问题。尽管一个非常高级别的可用性要求可能看起来很诱人，但大多数的系统的请求模式是间歇性的，（比如说一个商业系统最多使用的时间是在工作时间）而且单一的可用性数据不能真正反映用户的需求。在系统使用期间你将需要很高的可用性但是其他时候并不需要。当然，这依赖于系统的类型，可能现实当中 0.999 与 0.9999 并没有区别。

过描述的根本问题是可能在现实当中不可能证明达到一个很高的可靠性或可用性。比如，一个系统被设计来在安全性要求极高的应用中使用，并因此需要在其整个生命周期中永远不会失败。假设将会安装该系统的 1000 个拷贝，且系统每秒钟运行 1000 次。系统的预计生命期是 10 年。总的系统执行次数大约是 3×10^{14} 。描述失败发生率为执行次数的 $1/10^{15}$ 是没有意义的（这要考虑到一些安全性因素），因为你不能用足够长时间去测试系统来验证这个系统的这样的可靠性水平。

机构因此必须现实地对待是否值得去描述和验证一个高水平的可靠性系统的问题。高可靠性水平在那些可靠操作要求极高的系统中显然是合理的，比如电话交换系统，或系统失败可能导致大的经济损失的系统。但是对于很多种类的商业和科研系统就不是那么回事了。这样的系统有温和的可靠性需求，比如失败的损失仅仅是过程的耽搁并且从这些损失中恢复是简单的而且相对不昂贵的。

可以通过以下步骤避免对系统可靠性过度描述：

1. 针对不同类型的失败定义可用性和可靠性需求。严重失败的发生概率应该比小的失败发生的概率低。
2. 针对不同类型的服务定义可用性和可靠性需求。影响最紧要服务的失败发生概率要较低，只有局部影响的失败的发生概率可以相对较高。因而你要限制对影响最紧要系统服务的量化的可靠性描述。
3. 决定是否真正需要软件系统的高可靠性需求，或者是否系统可靠性目标可以通过其他方式达到。比如说，可以使用错误探测机制来检查系统的输出并有一个修改错误的过程。这样的话产生此输出的系统可能就不需要高可靠性了。

为解释第 3 个观点，考虑能够存取现金和为消费者提供其他服务的银行 ATM 机的可靠性需求。如果有硬件和软件的 ATM 问题，将导致非正确的输入到消费者账户数据库。这些可以通过定义 ATM 机一个非常高的硬件和软件可靠性得以避免。

然而，对于怎样识别和纠正不正确的账户交易银行拥有多年的经验。他们使用会计方法来检查什么时候出的错。大多数的事务如果失败了可以简单地把它取消掉，结果是不会引起银行的损失而且对于用户造成的不便也很小。运行 ATM 网络的银行因此接受这样的一个事实：即 ATM 的失败可能意味着很小一部分的交易是不正确的。但是事后再修改这些错误较之为了避免错误交易而做很高的花费要更经济得多。

对于一个银行来说（以及银行的用户），ATM 网络的可用性比个别 ATM 交易是否失败更重要。可用性的缺乏意味着对柜台服务发生更多请求、用户不满、修复网络的工程花费，等等。因此，对于基于事务的系统，比如银行业和电子商务系统，可靠性描述的焦点通常在于描述系统的

可用性。

为了定义一个 ATM 网络的可用性，你应该识别系统的服务，并且对每一项服务定义其所需的可用性需求。它们是：

- 客户的账户数据库服务；
- 单个 ATM 所提供的每一项服务，比如“取款”、“提供打印凭条”等。

这里，数据库的服务是最严格的，因为这项服务的失败意味着网络中的所有 ATM 都不能提供服务。因此，你应该把这项服务描述为高可用的。在这种情况下，早上 7 点到 11 点一个可以接受的可用性数值（忽略安排好的维护和升级）可能是 0.9999 左右。这意味着每周不可用的时间少于一分钟。现实当中，这将意味着很少的用户会受到影响而且会引起很小的用户不便。

对于每一个 ATM，总的可用性依赖于机械的可靠性和是否现金用尽。软件问题产生的影响应该比这些问题产生的影响要少。因此，ATM 的软件可用性级别相对不高是可以接受的。因此 ATM 软件的可用性可描述为 0.999，意味着在一天当中一台机器不可用的时间为 1 ~ 2 分钟。

作为可靠性描述的进一步的例子，考虑胰岛素注射系统。在这个系统中，每天要多次注射胰岛素，用户的血糖浓度每小时监视。因为系统的使用是间歇的，而且失败后果是严重的，最适当的可靠性度量是 POFOD（请求服务失败的概率）。

可以将胰岛素泵的失败分为两类：

1. 短暂的软件失败。这是一类可以由用户来修复的失败，比如，可以重启或重新校准机器。对于这种类型的失败，相对较低的 POFOD 值（假设 0.002）是可以接受的。这意味着失败可能在每 500 个请求中发生一次，大约为每 3.5 天一次，因为血糖含量每小时检查 5 次。

2. 永久性软件失败。机器需要由厂家维修。这个类型的失败概率非常低才行。大约一年一次是最低要求，因此，POFOD 应该不会高于 0.000 02。

然而，如下面一节所讨论的，胰岛素失败不会带来生命危险，所以在这里决定需求的可靠性级别是商业因素而非安全因素。由于用户需要一个非常快的维修和更换服务，所以服务的费用是非常高的。是厂商的利益决定了对需要修理的永久性失败的数量。

12.3.3 功能可靠性描述

功能可靠性描述包括识别影响系统的可靠性的约束和特征的需求。对于可靠性经过量化描述的系统，这些功能性需求对于确保达到可靠性所要求的水平是必须的。

系统中有三种功能性可靠性需求：

1. **检查需求** 这些需求识别对系统输入的检查，以确保不正确的或者是越界的输入在被系统处理之前被检测到。

2. **恢复需求** 设置这些需求来帮助系统在一次失败发生之后的恢复。典型的，这些需求关注的是维护系统及其数据的拷贝并定义在失败后如何重新修复系统服务。

3. **冗余性需求** 这些需求定义系统的冗余特征来保证一个组件的失败不会导致整个系统的失败。在下一章中会详细讨论这一点。

另外，可靠性需求可能包括可靠性过程需求。这些需求是用来确保在开发过程中使用那些好的实践经验，主要是能够降低系统失败的一些做法。图 12-8 给出了一些功能可靠性和过程需求的例子。

没有简单的规则可以导出功能性可靠性需求。在开发要求极高的系统的机构里，通常机构内部的关于可能的一些可靠性需求和这些需求是怎样在实际中影响系统的可靠性的经验。这

些机构可能对专门种类的系统特别在行，比如铁路控制系统，这样可靠性需求就可以在此类系统中复用了。

RR1：应该给出操作员的所有输入的预定义范围，系统应该检查所有操作员的输入是否落在此预定义的范围之内（检查）
RR2：病人数据库的拷贝应该保存在两个独立的服务器上，而这样的两个独立服务器是不能共存于同一座建筑物中的（恢复，冗余）
RR3：应该使用 N 版本编程来实现制动控制系统（冗余）
RR4：系统必须用 Ada 的一个子集来实现，并使用静态分析去检查（处理）

图 12-8 功能可靠性需求实例

12.4 信息安全性描述

系统的信息安全性需求描述与安全性需求有很多相似之处。对它们进行量化描述是不太现实的，信息安全性需求往往是“不应该”需求类型，它定义那些无法接受的系统行为，而不是定义系统功能。但是，信息安全性是比安全性更加有挑战性的属性，有以下几个原因：

1. 当考虑到安全性的时候，你可以认为系统所安装的环境不是有敌意的。没有人想要引起一个安全性相关的事故。当考虑到信息安全性的时候，你必须要假设对系统的攻击是故意的，并且系统的攻击者了解系统的弱点。

2. 当系统失败发生并带来安全性的风险时，你需要寻找导致失败产生的错误或者疏忽。当故意的攻击导致系统失败时，找到根本原因可能更加困难，因为攻击者可能会试图掩盖系统失败的原因。

3. 通常关闭系统或者降低系统服务可以避免安全性相关的失败，这些做法是可以接受的。然而，对系统的攻击可能是被称为拒绝服务的攻击，目的就是要关闭系统。关闭了系统就意味着攻击成功。

4. 安全相关事件不是由一个聪明的对手制造的。而一个攻击者可以在一系列攻击中试探系统的防御，在他更加了解系统以及系统的反应的时候可以修正攻击的方式。

这些区别意味着信息安全性需求成本一定要比安全性需求成本高。安全性需求导致产生功能性系统需求，提供对可能引发系统安全性相关失败的事件和缺陷的保护。它们更多关注检查问题并在问题发生时采取行动。相反地，有很多种信息安全性相关的需求，覆盖系统所面临的各种威胁。Firesmith (Firesmith, 2003) 找出了包含在系统中的 10 类信息安全需求：

1. 身份验证需求定义系统是否应该在用户与之交互之前辨认其身份；
2. 认证需求定义系统如何验证用户身份；
3. 授权需求定义对合法用户的权利和访问许可；
4. 免疫需求定义系统如何保护自己免受病毒、蠕虫以及类似威胁的入侵；
5. 完整性需求定义如何避免数据崩溃；
6. 入侵保护需求定义在系统中应该使用什么机制来检测对系统的攻击；
7. 不可抵赖需求定义参与交易的一方不能对自己已经作出的交易抵赖；
8. 隐私需求定义如何维护数据的私密性；
9. 信息安全的审查需求定义如何对系统的使用进行审计和检查。
10. 系统维护的信息安全需求定义应用如何避免因信息安全机制的意外失败所导致的授权修改。



信息安全风险管理

安全性是个法律问题，商业一定要生成安全的系统。然而，信息安全的某些方面是商业问题，即商业可以决定不去实现某些信息安全措施并去遮掩由此所带来的损失。风险管理是决断哪些资产必须得到保护以及可以在这上面花费多少的过程。

<http://www.SoftwareEngineering-9.com/Web/Security/RiskMan.html>

当然，你不会在每个系统当中看到所有这些种类的需求。特别的需求依赖于特别的系统、特别的使用环境，以及特定的用户。

12.1 节讨论的风险分析和评估过程可以用在验证系统信息安全性需求上。正如已经讨论过的，这个过程有 3 个步骤：

1. **初步的风险分析** 在这个阶段，关于系统需求、系统设计，或是实现技术的详细内容上的决定还没有做出。这个评估过程的目标在于导出系统的整体信息安全性需求。

2. **生命周期风险分析** 这个风险评估发生在设计选择已经做出之后的系统开发周期内。附加的信息安全性需求考虑到了构建系统、系统设计以及实现决策中所使用的技术等因素。

3. **运行风险分析** 这个风险评估考虑在用户的恶性攻击运行的系统所带来的风险，无论这些用户是否有系统内部信息。

信息安全性需求描述中使用的风险评估和过程分析是 12.1 节所讲的一般的风险驱动描述过程的变种。一个风险驱动的信息安全性需求过程在图 12-9 中给出。看起来它可能与图 12-1 中所示的风险驱动过程不同，但是指出每个阶段是如何对应一般过程中的某个阶段的（在下面的说明中，括号中写出的是所对应的一般性过程中的活动）。此过程中的阶段有：

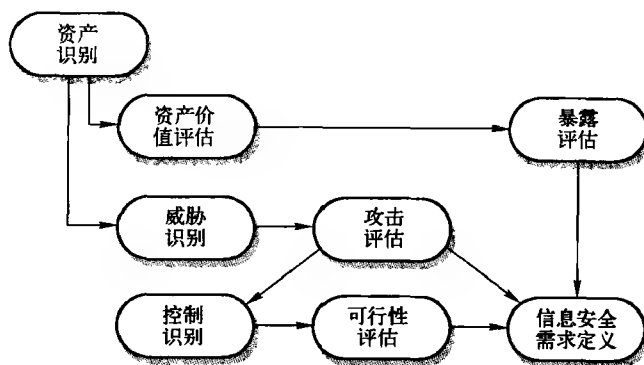


图 12-9 信息安全需求中的初步风险评估过程

1. **资产识别**，识别出可能需要保护的系统资产。系统自身或者是特殊的系统功能可能以及系统相关的数据都是资产（风险识别）。

2. **资产价值评估**，估计你所识别的资产的价值（风险分析）。

3. **暴露评估**，也就是评估与每一份资产相关联的潜在损失。这个步骤应该考虑到如信息失窃一类的直接损失，恢复的花费，以及可能的名誉上的损失（风险分析）。

4. **威胁识别**，就是识别对系统资产的每一项威胁（风险分析）。

5. **攻击评估**，需要把每项威胁分解为可能对系统的各个攻击，并给出这些攻击会以什么方式进行。可以使用攻击树（Schneier, 1999）来分析可能的攻击。这与缺陷树相类似，应该在树根的部分以一个威胁开始，并识别可能的攻击以及这些攻击将会是怎样造成的（风险分解）。

6. **控制识别**，提出你为保护资产而可能使用的控制方式。这些控制是一些技术机制以用来

保护资产，比如加密（风险降低）。

7. 可行性评估，评估技术的可行性以及所提出的控制的开销。使用昂贵开销的控制来保护价值不高的资产是不值得的（风险降低）。

8. 信息安全性需求定义，使用关于暴露、威胁和控制评估的知识去导出系统信息安全性需求。它们可能是系统基础结构的需求或是应用系统的需求（风险降低）。

风险评估和管理过程的一个重要输入是机构的信息安全政策。一个机构的信息安全政策是面向所有系统的，并且规定了什么是允许的和什么是不能允许的。比如，军事信息安全性政策的一个方面可以被陈述为：“阅读者仅可以查阅那些文档等级与阅读者的审查等级相同或者低于阅读者的审查等级的文档。”这意味着如果一个阅读者被检查后设定级别为“绝密”，他们可以访问标为“绝密”、“机密”或者“公开”的文档，但是不可以访问标记为“最高机密”的文档。

信息安全政策阐明什么是信息安全的系统所必须维持的，并因此帮助识别可能产生的威胁的那些条件。所谓威胁是指可以威胁商业秘密的任何东西。现实当中，信息安全性政策通常是定义什么是允许的什么是不允许的非正式的文档。然而，Bishop（2005）讨论了用正规语言表达信息安全性政策并生成自动的检查来保证政策得到遵守。

为说明这个信息安全性风险分析的过程，考虑医院里的心理健康治疗信息系统 MHC-PMS。这里没有足够的篇幅系统地讨论风险评估，只是利用这个系统作为例子。其报告的片段如图 12-10 和图 12-11 所示。这个初步的风险分析报告用于定义信息安全性需求。

从医院的信息系统的风险分析报告中可以获得信息安全性需求。这些需求中的一些例子是：

- 1. 在诊断阶段的开始，病人信息必须被下载，从数据库下载到系统客户端的一个安全区域。
- 2. 系统客户端中的所有的病人信息应该被加密。
- 3. 当诊断阶段结束时，应该把所有的病人信息都上传到数据库并删除系统客户端的备份。
- 4. 对系统数据库的所有修改以及这些修改的创建者信息应该记录成日志并保留在不同于数据库服务器的另一台计算机上。

资 产	价 值	暴 露
信息系统	高，需要用于支持所有诊所咨询。 潜在的安全要求极高	高。诊所会被注销营业执照。恢复系统的成本。如果不能得到及时治疗，患者可能受到的伤害
病人数据库	高，需要用于支持所有诊所咨询。 潜在的安全要求极高	高。诊所会被注销营业执照。恢复系统的成本。如果没有得到及时治疗，患者可能受到的伤害
单条病人记录	通常低，尽管对于专门的身份显赫的病人是高的	低。但可能带来声誉上的损失

图 12-10 MHC-PMS 初步风险评估报告中的资产分析

威 胁	可 能 性	控 制	可 行 性
未经授权的用户以系统管理员的身份获得访问权限，并使得系统不再可用	低	只允许从某个物理上是安全的专门位置进行系统管理	低实现费用，但必须看管好密钥，并且确保在紧急时刻能拿到密钥
未授权的用户以系统用户身份访问系统，并获得私密信息	高	要求所有用户使用生物机制进行身份验证，对病人信息的所有修改记录日志以跟踪系统的使用	技术上是可行的，但是这是一种很高代价的解决方案。可能引起用户的抵制。实现是简单的透明的，也是支持恢复的

图 12-11 在初始风险的评估报告中的威胁和控制分析

前两条需求是相关的——病人的信息被下载到本地机器，这样即使病人数据库服务器受到

攻击或者变得不可用，会诊也可以继续。然而，必须删除这些信息，这样后来的客户端电脑使用者才不能访问这些数据。第四个需求是一个恢复和审计需求。它意味着这些变化可以通过重现修改日志被恢复并且可能发现是谁做出的修改。这个机制可以阻止授权员工对系统的误用。

12.5 形式化描述

30 多年来，很多研究者提出了使用形式化方法来进行软件开发。形式化方法是基于数学的软件开发方法，通过定义软件的一个模型，接着可以形式化地分析这些模型，并使用它作为一个形式化系统描述的基础。原则上讲，从形式化模型开始构造软件，并证明所开发的程序与这个模型是一致的，以此来消除由于程序错误造成的软件失败。这是可能的。



形式化描述技术

形式化系统描述可以采用两种基本方法表达，一是用系统界面模型（代数描述），二是用系统状态模型。读者可以下载一个额外的网上关于此话题的一章内容，在那儿作者给出了两种方法的例子。在那一章中包含胰岛素泵系统的一部分的形式化描述。

<http://www.SoftwareEngineering-9.com/Web/ExtraChaps/FormalSpec.pdf>

所有形式化开发过程都开始于形式化系统模型，把它作为系统的描述。为了建立这个模型，你需要将自然的语言、图和表格等表述的用户需求翻译成一种数学语言描述的形式化定义的语义。形式化描述是对系统应该做什么的一个无二义的描述。使用手工的或者借助工具的方法，你可以检查程序的行为是与描述一致的。

形式化描述对于设计的证明和软件的实现不仅仅是必不可少的，它们还是定义系统最精确的方式，所以减少了误解的可能。另外，构造一个形式化的描述强制需要一个细致的需求分析，而这又是一个发现需求存在的问题的有效方法。在一个自然语言描述中，错误可能被不精确的语言所掩盖，而在形式化的描述过的系统中不会出现这样的问题。

形式化描述总是作为计划驱动的软件过程的一个部分。在这样的软件过程中系统需求完全是在开发之前定义好了的。系统需求和设计经过了详细的分析和检查，在实现环节开始之前进行了详细定义。如果要开发软件的形式化描述，一般总是在系统需求定义之后，系统详细设计开始之前进行。在详细需求描述和形式化描述之间有一个紧密的反馈回路。

图 12-12 给出了基于计划的软件过程中的软件描述各个阶段以及它们与软件设计的接口。由于开发形式化描述是很昂贵的，所以你可以决定将此方法局限于那些对于系统运行十分关键的部分组件的构造上。因而需要我们在系统体系结构设计的时候识别出这些组件。

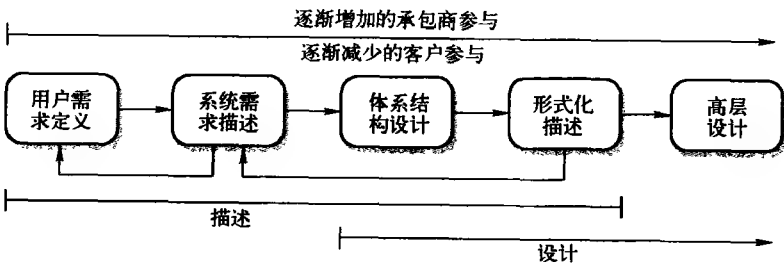


图 12-12 在基于计划的软件过程中的形式化描述

在过去的几年时间里,分析形式化描述的自动支持已经得到很大发展。模型检查器(Clark等,2000)是一种软件工具,它是将基于状态的形式化描述(一种系统模型)以及某些形式化表达所希望的性质作为输入,例如,“不存在不可到达的状态。”模型检查程序全面地分析描述,并报告系统性质能够得到满足,或者给出例子说明性质无法得到满足。模型检查与静态分析是紧密关联的,第15章将讨论这些系统验证的一般方法。

开发形式化描述并将其用于一个形式化开发过程的优势是:

1. 当你详细地开发一个形式化描述的时候,你获得了对系统需求的深度的细致的理解。就算你并不是将它用于形式化开发过程,开发形式化描述也是需求错误检测的一个有说服力的理由(Hall,1990)。改正早期被发现的需求问题通常比在晚一些的阶段的开发过程中修改它代价相对要小很多。

2. 因为描述是用一种带有形式定义的语义的语言来表达的,你可以自动地分析它来发现其中的不一致和不完整之处。

3. 如果你使用一个方法(比如B方法),你可以使用一系列正确性保持变换将形式化描述转换成程序。这样其结果程序可以保证与描述是一致的。

4. 程序检测的花费可能会降低,因为你已经根据描述验证了程序。

尽管有这些长处,形式化描述还是在现实的软件开发当中受到很大的限制,即便对于要求极高的系统也是这样。所以,在开发和使用形式化描述方面积累的经验是非常少的。反对开发形式化描述的理由如下:

1. 提出问题的人和领域专家不能理解形式化描述,所以他们不能检查形式化描述是否能表达他们的需求。软件工程师理解形式化描述,但是却可能不了解应用领域,所以他们同样不能确定形式化描述是否是系统需求的精确反映。

2. 对创建一个形式化描述的花费是很容易量化的,但是评估因使用它所带来的成本节余就要困难得多。结果,管理者不愿意冒风险来采用这个方法。

3. 多数的软件工程师没有受过形式化描述语言的训练。这样,他们也就不愿意在开发过程中提议用这个方法。

4. 很难将现有方法扩展到一个很大的系统中。当使用形式化描述时,最通常的做法是只对要求极高的核心部分进行形式描述,而不是对整个系统使用此方法。

5. 形式化描述不适用于敏捷方法开发。



形式化描述的成本

开发形式化描述是个花费很大的过程,因为需要很多时间来将需求转换成形式语言,并检查此描述。经验表明,采用形式化描述可以在系统测试和检验方面带来很大节省,因而看来形式化地描述系统不会带来总的开发成本的上涨。然而,开发成本的分布发生了改变,在早期的开发过程中占用了更多的成本。

<http://www.SoftwareEngineering-9.com/Web/FormalSpecCosts/>

尽管如此,在写这本书的时候形式化描述还是被用在开发一定数量的安全 and 信息安全要求极高的系统中。它们可能同样用于在开发和验证较大的并且复杂的软件系统中那些要求极高的部件上,在这样的开发中形式化方法是很划算的(Badeau和Amelot,2005;Hall,1996;Hall和Chapman,2002;Miller等,2005;Wprdwprth,1996)。它们是要要求极高系统工程中静态检验工具的基础,这样的静态检验工具如微软的驱动检验系统(Ball等,2004;Bell等,2006)和

SPARK/Ada 语言 (Barnes, 2003)。

要点

- 危险分析在安全性描述过程中是一个主要活动。它包括识别危险的状态, 这些状态能引起系统的危险。系统需求就是要保证这些危险不会出现, 如果危险一旦发生, 保证不会造成事故。
- 一个危险驱动的方法可能用在理解系统的安全性需求上。你要找出潜在的危险并且化解这些危险 (使用例如缺陷树分析的方法) 来发现它们的根本原因。然后你要描述这些需求来避免出现问题或者从问题中恢复。
- 可靠性需求应该在系统需求描述中量化定义。可靠性量度包括请求失败的可能性 (POFOD)、失败发生的频率 (ROCOF) 以及可用性 (AVAIL)。
- 不要过描述系统可靠性需求是很重要的, 因为这会导致开发阶段和验证过程中的额外花费。
- 信息安全性需求比安全性需求更加难以识别, 因为一个系统攻击者可以使用系统脆弱性知识来计划一次系统攻击, 并且可以从不成功的攻击当中学到很多系统脆弱性知识。
- 为定义信息安全性需求, 应该找出要保护的资产, 并定义应该采取什么样的信息安全技术来保护它们。
- 软件开发的形式化方法依赖于表达为数学模型形式的系统描述。开发一个形式化描述对于促进对系统需求的细致检查和分析有重大好处。

进一步阅读材料

《Software: System Safety and Computers》, 是一本针对安全性要求极高的系统各个方面展开彻底讨论的书。它的长处在于对危险分析的阐述和从这些分析中导出需求 (N. Leveson, 1995, Addison-Wesley)。

《Security Use Case》, 这是一篇很好的论文, 可以在网上看到。主要研究的是如何在信息安全描述中使用用例。该论文的参考文献中还列出了该文作者在信息安全描述方面的其他一些好文章 (D. G. Firesmith, Journal of Object Technology, 2(3), May-June 2003)。http://www.jot.fm/issue_2003_05/column6/。

《Ten Commandments of Formal Methods... Ten Years Later》给出了一套使用形式化方法的指南, 这些指南于 1996 年首次被推荐并且再次在本书中出现。它是围绕形式化方法使用的实际问题的一个很好的总结 (J. P. Bowen and M. G. Hinchey, IEEE Computer, 39(1), January 2006)。http://dx.doi.org/10.1109/MC.2006.35。

《Security Requirements for the Rest of Us: A Survey》一篇很好的信息安全性需求描述的初学者读物。作者更加关注轻量级方法而不是形式化方法 (I. A. Tondel, M. G. Jaatun, and P. H. Meland, IEEE Software, 25(1), January/February 2008)。http://dx.doi.org/10.1109/MS.2008.19。

练习

- 12.1 解释为什么在图 9-2 所示的风险三角形的边界容易随着时间的推移和变化了的社会意识而发生改变?
- 12.2 解释为什么在描述安全性和信息安全性的时候基于风险的方法解释成不同的方式。
- 12.3 在胰岛素注射系统中, 用户需要定期更换针头和胰岛素的供给, 还会改变单个剂量的最大

值和日剂量的最大值。提出 3 个可能发生的用户错误，并给出安全需求来避免这些错误造成事故。

- 12.4 一个治疗癌症患者的安全性要求极高的软件系统有两个主要的组件：

- 一个射线治疗仪，能对肿瘤部位进行辐射治疗，辐射剂量能够控制，这个仪器是由一个嵌入式软件系统控制的。
- 一个治疗数据库，包括对每位病人的治疗的详细情况。治疗需求被输入到这个数据库中，并自动输出到辐射治疗仪中。

识别出 3 个可能出现在系统中的危险。对每个危险，提出能减少危险造成事故可能性的防范需求。解释为什么你提议的防范措施能减少危险的发生。

- 12.5 为下列软件系统提出合适的可靠性度量。为你的选择给出合适的理由。对这些系统的用法给出预测，并给出这些可靠性度量的合适的值：

- 医院特护病房中监控病人的系统；
- 字处理器；
- 自动售卖机控制系统；
- 汽车刹车控制系统；
- 制冷部件控制系统；
- 管理报告生成器。

- 12.6 如果火车在某个路段上的时速超过了限制或者是火车在进入某个红灯亮的路段（例如，该路段不允许驶入）的时候，火车的保护系统就会自动工作。选择用于描述这个系统所要求的可靠性的可靠性量度，为你的答案给出理由。

- 12.7 对于第 12.6 题这样的系统，有两个主要的安全性需求：

- 火车不能够驶入红灯亮的路段；
- 火车速度不应当达到该路段所规定的速度极限。

假设某路段的信号状态和速度极限在机车进入该路段之前被传输到火车上的软件中，该软件将产生于系统安全性需求，请为该车载软件提出 5 种可能的功能性系统需求。

- 12.8 解释为什么在系统开发中需要初步的信息安全风险和生存周期信息安全风险评估。

- 12.9 扩展图 12-11 中的表格再多识别出两个 MHC-PMS 的威胁，以及相关的控制。以它们为基础产生进一步的信息安全性需求来实现所提出的控制。

- 12.10 从事安全性相关系统的需求描述和开发工作的软件工程师需要某种专业认证吗？解释你的理由。

参考书目

Badeau, F. and Amelot, A. (2005). 'Using B as a High Level Programming Language in an Industrial Project: Roissy VAL'. Proc. ZB 2005: Formal Specification and Development in Z and B, Guildford, UK: Springer.

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K. and Ustuner, A. (2006). 'Thorough Static Analysis of Device Drivers'. Proc. EuroSys 2006, Leuven, Belgium.

Ball, T., Cook, B., Levin, V. and Rajamani, S. K. (2004). 'SLAM and Static Driver Verifier: Technology Transfer of Formal Methods Inside Microsoft'. Proc. Integrated Formal Methods 2004, Canterbury, UK: Springer.

- Barnes, J. P. (2003). *High-integrity Software: The SPARK Approach to Safety and Security*. Harlow, UK: Addison-Wesley.
- Bishop, M. (2005). *Introduction to Computer Security*. Boston: Addison-Wesley.
- Brazendale, J. and Bell, R. (1994). 'Safety-related control and protection systems: standards update'. *IEE Computing and Control Engineering J.*, 5 (1), 6–12.
- Clarke, E. M., Grumberg, O. and Peled, D. A. (2000). *Model Checking*. Cambridge, Mass.: MIT Press.
- Firesmith, D. G. (2003). 'Engineering Security Requirements'. *Journal of Object Technology*, 2 (1), 53–68.
- Hall, A. (1990). 'Seven Myths of Formal Methods'. *IEEE Software*, 7 (5), 11–20.
- Hall, A. (1996). 'Using Formal methods to Develop an ATC Information System'. *IEEE Software*, 13 (2), 66–76.
- Hall, A. and Chapman, R. (2002). 'Correctness by Construction: Developing a Commercially Secure System'. *IEEE Software*, 19 (1), 18–25.
- Jahanian, F. and Mok, A. K. (1986). 'Safety analysis of timing properties in real-time systems'. *IEEE Trans.on Software Engineering.*, SE-12 (9), 890–904.
- Leveson, N. and Stolzy, J. (1987). 'Safety analysis using Petri nets'. *IEEE Transactions on Software Engineering*, 13 (3), 386–397.
- Leveson, N. G. (1995). *Safeware: System Safety and Computers*. Reading, Mass.: Addison-Wesley.
- Miller, S. P., Anderson, E. A., Wagner, L. G., Whalen, M. W. and Heimdahl, M. P. E. (2005). 'Formal Verification of Flight Control Software'. *Proc. AIAA Guidance, Navigation and Control Conference*, San Francisco.
- Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. New York: McGraw-Hill.
- Schneier, B. (1999). 'Attack Trees'. *Dr Dobbs Journal*, 24 (12), 1–9.
- Storey, N. (1996). *Safety-Critical Computer Systems*. Harlow, UK: Addison-Wesley.
- Wordsworth, J. (1996). *Software Engineering with B*. Wokingham: Addison-Wesley.

可依赖性工程

目标

这一章的目标是讨论开发高可依赖性系统的过程和技术。读完本章，你将了解以下内容：

- 理解系统的可依赖性是怎么通过使用冗余的和多样的组件而实现的；
- 知道可依赖性软件过程是怎样支持可依赖性软件开发的；
- 理解可能使用的不同的系统体系结构风格来实现软件的冗余性和多样性；
- 了解应该在可依赖性系统工程中使用的好的编程实践。

软件工程技术的使用、更好的编程语言和更好质量的管理使得大多数软件的可依赖性得到了显著的提高。然而，系统失败可能依然会发生，从而影响系统的可用性或者导致错误的结果产生。在有些情况下，这些失败仅仅产生很小的不便。系统拥有者可能只是决定不修改系统的错误而放任这些失败的存在。然而在有些系统中，失败可能导致生命损失和重大的经济或名誉损失。这些就是所谓的“要求极高的系统”，因为每一个高级别的可依赖性都是必须达到的。

要求极高的系统的例子包括：过程控制系统，在失败事件中关闭其他系统的保护性系统，医疗系统，电话交换机和飞行控制系统。特殊的软件开发工具和技术可能用在要求极高的系统当中来增强软件的可依赖性。这些工具和技术通常增加了系统开发的花费，但是它们降低了系统失败的风险以及失败可能造成的损失。

可依赖性工程是关于在要求极高和非要求极高的系统中所采用的技术。这些技术支持 3 个用在开发可依赖性软件方面的补充方法：

1. **缺陷避免** 在系统的设计和实现过程中使用一些软件开发方法来减少编程缺陷发生，并在系统投入使用之前发现系统中的缺陷，这些缺陷在系统执行期间可能发生。更少的缺陷意味着更少的运行时失败的机会。

2. **缺陷检测** 在部署使用之前设计检验和有效性验证过程来发现和去除程序中的缺陷。要求极高的系统需要非常昂贵的检验和有效性验证，以便在部署之前发现尽可能多的缺陷并且令系统的拥有者确信系统是可靠的。我们将在第 15 章叙述这个内容。

3. **容错** 系统应该设计成这样：系统的缺陷及无法预期的系统行为在执行时能够得到检测和管理，避免导致系统失败。在所有的系统中都应该包含一种基于内嵌的运行时检测的容错方法。可是更多的特殊的容错技术（比如容错系统体系结构的使用）通常来说只用在需要非常高级别的系统可用性和可依赖性的时候。

不幸的是，应用错误避免、错误检测和容错技术导致出现了一个投入收益递减的状态。发现和清除软件系统程序中的残余缺陷的成本呈指数增加（见图 13-1）。当软件越来越可依赖时，用于查找越来越少的缺陷的时间和精力将会越来越多。在某个时期，这些额外花费是不合理的。

结果，软件开发公司默认他们的软件存在一些残余缺陷。允许缺陷的水平依赖于系统类型。塑封产品允许相对较高的缺陷水平，而要求极高的系统通常要求较低的缺陷密度。

可以接受的缺陷的基本原则是，当系统失败时，失败后果造成的损失比在系统发布之前发现和删除它们的费用要少。但是，就如第 11 章讨论的，发行有缺陷软件并不仅仅是一个经济问题，同时还要考虑系统失败的社会和政治的可接受性。

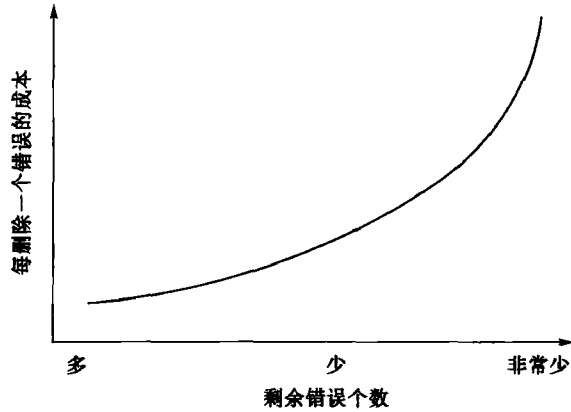


图 13-1 去除驻留缺陷的费用在不断攀升

很多要求极高的系统，比如飞行控制系统、医药系统、会计系统等，被用在规范的领域，比如航空运输、医药和金融。国家政府定义这些行业中的应用规范并且制定一个监管部门来保证这些公司都遵守这些规范。现实当中，这意味着监管者经常必须确信要求极高的软件是可以被信任的，而且这需要清晰的证据证明这些系统是可靠的。

因此，要求极高系统的开发过程不仅仅是关于制造一个可靠的系统的问题；它必须同时制造出证据来让监管者相信系统是可靠的。制造这样的一个证据将花费掉很大比例的开发费用，所以它是要求极高的系统很昂贵的重要原因之一。第 15 章将讨论安全性和可靠性用例的问题。



阿丽亚娜 5 号火箭爆炸

1996 年，欧洲航天局的阿丽亚娜 5 号火箭在其首次飞行开始 37 秒后发生爆炸。事件是由于软件系统失败造成的。火箭上有个软件备份系统，但这个系统并不具有多样性，所以备份计算机也以同样的方式失败了。火箭和它的卫星的有效载荷全部被摧毁。

[http://www. SoftwareEngineering-9. com/ Web/ DependabilityEng/ Ariane/](http://www.SoftwareEngineering-9.com/Web/DependabilityEng/Ariane/)

13.1 冗余性和多样性

冗余性和多样性是增强任何类型系统的可靠性的基本策略。冗余性意味着系统中包含多余的能力可以在系统失败的时候发生作用。多样性意味着冗余的系统组件是属于不同种类的，这样就会增加它们不会以完全一样的方式失败的机会。

我们使用冗余性和多样性来增加我们日常生活中的可靠性。一个冗余性的例子是，多数人在家里有备用的灯泡，这样可以快速从正在使用的灯泡的失败中恢复。通常，为保证我们的居所安全我们通常使用几道锁（冗余性），并且，通常我们使用不同种类的锁（多样性）。这意味着如果一个侵入者找到破坏其中一个锁的方法，他们还必须找到不同的方法来破坏其他的锁才能进入。如日常工作一样，我们应该完全备份我们的电脑中的内容并保持我们的数据的多个冗余拷贝。为避免磁盘失败的问题，备份应该在单独的、不同的外部设备上被保存。

设计用来增强可靠性的软件系统可能包括提供与系统中的其他组件相同功能的冗余组件。

这些组件在主组件失败的时候被选择加入系统。如果这些冗余组件是多样的（比如，与其他组件不同），一个普通的缺陷不会引起一个有备份组件的系统的失败。冗余性也可以通过包含附加的检查代码来提供，这些代码可能并不是直接对系统功能有用。可以在一些缺陷导致失败之前探测到这些代码。它们可以激发恢复机制工作以保证系统继续运行。

在可用性要求高的系统中，冗余的服务常常被使用。这些服务在指定的服务失败的时候自动地运行。有时候，为保证攻击不能够暴露一个常见的弱点，这些服务可能属于不同的类型并可能在不同的操作系统上运行。使用不同的操作系统是软件多样性和冗余性的例子之一，并行的功能以不同的方法实现。13.3.4节将讨论软件多样性的更多细节。

多样性和冗余性可能同样用于获得一个可依赖的过程，这是通过确保过程活动（比如软件的有效性验证）不依赖于单个过程或方法而实现的。这增强了软件的可依赖性，因为它降低了过程失败的机会，也就是由于在软件开发过程中人的错误导致的软件错误。比如说，有效性验证活动可能将程序测试、手工程序检查以及静态分析作为缺陷查找技术。这些是互补的技术，因为任何一个技术都可能找到其他技术找不到的缺陷。另外，不同的组员可能负责一些活动（比如，程序检查）。人们用何种方法处理工作取决于他们的个性、经验和受教育程度，所以这种冗余性提供了对系统不同的视角。



可依赖的操作过程

本章讨论可依赖的开发过程，但是对于系统的可依赖性还有一个同样重要的方面，那就是系统的运行过程。在设计这些系统的操作过程中，你需要考虑到人的因素，并总是在心中牢记人在使用系统时是非常容易出错的。设计一个可依赖的过程应该避免人为错误，当错误形成时，软件应该能检测到此错误并允许使用者去改正错误。

<http://www.SoftwareEngineering-9.com/Web/DependabilityEng/HumanFactors/>

如13.3.4节介绍的那样，获得软件多样性不是简单的。多样性和冗余性使得系统更加复杂并且通常难以理解。不仅仅因为需要编写更多的代码并检查它们，还需要有附加的功能加入到系统中来以探测组件的失败并切换控制到候选组件。这些额外的复杂性使得程序员更容易犯错误，且系统检查人员更难以检查到错误。

结果是，一些人认为最好避免软件冗余性和多样性。他们的观点是设计软件要设计得越简单越好，可以使用极为严格的软件检验和有效性验证过程（Parnas等，1990）。更多的花费可以用在检验和有效性验证上，因为节省了开发冗余系统组件的花费。

两个方法都被用在商业上的安全极高的系统中。比如，空客340飞行控制系统硬件和软件既多样又冗余（Storey，1996）。波音777的飞行控制软件是基于冗余软件，每个计算机都运行同样的软件，这个软件是得到极度严格的验证的。波音777飞行控制系统把注意力集中在简单性而不是冗余性上。这两个飞行控制系统都是可依赖的，所以多样性和简单方法对于实现可依赖性来说都很明显是成功的。

13.2 可依赖的过程

可依赖软件过程是被用来开发可依赖软件的软件过程。公司使用了可依赖过程可以保证这个过程能得到正确执行和文档化，且适当的开发技术用在了要求极高的系统开发中。在可依赖过程上的投资的道理是，一个好的软件过程可以使所交付的软件包含更少的错误，并因此有更少的执行时失败。图13-2显示了一些可依赖软件过程的属性。

过程特点	描 述
可文档化	过程应该有一个定义好的过程模型，给出过程中的活动以及要在活动中产生的各种文档
标准化	应该有一个全面的覆盖软件开发和文档化的一组软件开发标准
可审查性	过程应该是能够被外部人员审查的。外部人员会审查过程标准是否得到遵守并给出过程改善的若干意见和建议
多样性	过程应该包括冗余和多样性检验以及有效性验证活动
鲁棒性	过程应该能够从单个过程活动的失败中恢复

图 13-2 可依赖过程的属性

对于外部管理者来说，应用可依赖过程的证据能令他们相信在软件开发中已经使用了最有效的软件工程实践。系统开发人员也将定期拿出一个过程模型以及过程被遵循的证据给管理者看。要使管理者相信所有参与者都一致地使用这个过程，并且该过程是可以被用在不同的开发项目中。这意味着过程必须被明确地定义并且是可重现的。

1. 明确定义的过程就是用已定义的过程模型来驱动软件生产过程。在过程中必须有数据收集说明过程模型中所有必要的步骤都得到了执行。

2. 可重复的过程是一个不依赖于个别解释和判断的过程。此过程可以在各个项目中重复使用并且在不同的团队人员中使用，不论谁在开发团队中。这对于要求极高的系统是特别重要的，因为这样的系统通常有一个很长的开发周期，并且在这个周期中开发小组的成员会经常发生变动。

可依赖过程利用冗余性和多样性来获得可靠性。可依赖过程经常包含相同目标的不同活动。比如说，程序审查和测试的目标是发现程序中的错误。这些方法是相互补充的，这样它们一起工作可能比只使用一种技术能发现更大比例的错误。

这些活动用在可依赖性过程中显然依赖于所开发的软件的种类。然而总的来说，这些活动应该相互协调以防止引入新的错误到系统中，并且探测并移除错误，保护过程本身的信息。在一个可依赖过程中可能包括的活动的例子有：



安全性生命周期

国际电子技术委员会针对保护性系统这类工程设计了一个过程标准（IEC 61508）。这是一个基于安全性生命周期概念的。它将安全性工程和系统工程明确地区分开来。IEC 611508 安全性生命周期的第一阶段定义了系统的范围，评估潜在的系统危险，估计会形成的风险。接下来是对安全性需求的描述并分配这些需求到不同的子系统。其思想是限制安全要求极高的功能的范围，以允许针对要求极高的系统工程的专门技术能应用到安全性要求极高的系统开发中来。

[http://www. SoftwareEngineering-9. com/ Web/ SafetyLifeCycle/](http://www.SoftwareEngineering-9.com/Web/SafetyLifeCycle/)

1. 需求复查以尽可能地检查需求的完备性和一致性。

2. 需求管理，保证需求的变更是受控制的，并且所有的受到变更影响的程序员都能够了解所提出的需求变更的影响。

3. 形式化描述，创建并且分析软件的数学模型。在第 12 章讨论了形式化描述的益处。可能它最重要的益处是它对系统需求施加一个非常详细的分析。这个分析本身可以用来发现可能在需求复查阶段被忽略的需求问题。

4. 系统建模，也就是用一组图形化的模型清晰地记录软件设计，并且需求和这些模型之间

的联系也被清晰地记录下来。

5. 设计和程序审查, 让不同的人分别检查系统不同的描述部分。通常是依据普遍性的设计和编程错误清单进行审查的。

6. 静态分析, 就是对程序的源代码进行自动检查。查找那些可能指示程序错误或疏忽的异常。我们将在第15章讨论静态分析。

7. 测试规划和管理, 就是设计一系列全面的系统测试。必须很小心地管理测试过程来证明这些测试已经覆盖了系统需求并且在测试过程中得到正确的应用。

如同聚焦系统开发和测试的过程活动, 我们还必须有完善定义的质量管理和变更管理过程。尽管一家公司的可依赖性过程中的特殊活动可能与另一家公司不同, 对有效的质量和变更管理的需要是一致的。

质量管理过程(将在第24章讨论)建立了一系列过程和产品标准。它们还包括捕捉过程信息的活动以证明这些标准被遵循了。比如说, 可能存在一个标准, 它被定义来执行程序审查。审查小组的组长负责记录过程以说明审查标准一直是得到遵循的。

在第25章讨论的变更管理是关于管理系统的变更, 确保一个接受的变更能够真正实现, 并肯定所计划的发行版本中包含了这些计划中的变更。一个通常出现的软件问题是在一个系统中包含了一个错误的组件。这可以导致一种情况的发生, 即一个执行系统中包含在开发过程中未经检查的组件。配置管理过程必须定义为变更管理过程的一部分, 以保证此类事件不会发生。

有一个普遍接受的观点是敏捷方法, 正如在第3章所介绍的, 并不是很适合可依赖性过程(Boehm, 2002)。敏捷方法专注于软件开发而不关心如何将已经做好的东西写成文档。敏捷方法经常用相对非正式的方法来实现变更和质量管理。用基于计划的方法来开发可依赖系统时, 创建文档能让外部管理者和其他外部信息持有者都可以理解, 通常是首选方法。尽管如此, 敏捷方法的益处同样可以适用于要求极高的系统中。有过在这个领域中成功应用敏捷方法的报告(Lindvall 等, 2004), 而且很可能将会开发出多种适合要求极高的系统工程的敏捷方法。

13.3 可依赖的系统体系结构

如已经讨论过的, 可依赖性系统开发应该基于一个可依赖性过程。然而, 尽管你可能需要一个可依赖性过程来创建可依赖系统, 其本身并不足以保证可依赖性。你同样需要为可依赖性设计一个系统体系结构, 尤其是当需要容错机制的时候。这意味着系统体系结构必须设计成包含冗余组件和机制, 能够允许控制可以从一个组件到另一个组件进行切换。

系统可能需要容错系统体系结构的例子包括: 飞行器系统在飞行期间必须可操作, 电信系统和要求极高的指控系统。Pullum (2001) 描述了人们提出的不同种类的容错系统体系结构, 而TorresPomales (2000) 综述了软件容错技术。

可依赖性系统体系结构的最简单的实现是带复制的服务器(备份服务器), 两个或多个服务器完成同样的任务。处理的请求通过一个服务器管理组件被路由到一个特定的服务器上。这个服务器管理组件同时还跟踪服务器响应。如果服务器失败, 这通常是通过无法得到响应而被探测到, 有缺陷的服务被系统剔除出去。未处理的请求由其他服务器重新接受并处理。

这种带复制的服务器方法广泛使用在事务处理系统中, 很容易维护要处理的事务处理的多个拷贝。事务处理系统的设计使得数据仅仅在事务正确结束才得到更新, 这样处理延迟不会影响系统的完整性。如果备用服务器被用在低优先权任务中, 那么这个方法可能是使用硬件的一个很有效的方法。如果主要的服务器发生了问题, 它的处理将转移到备用服务器进行, 该服务器对此任务给予最高优先权。

带复制的服务器提供冗余性, 但是并不经常提供多样性。硬件通常是同一的并且备份服务

器运行相同版本的软件。因此它们可以应对集中在一台机器上的硬件失败和软件失败。它们不能应对导致所有版本软件同时失败的软件设计问题。为应对软件设计的失败，一个系统必须拥有多样的软件和硬件，如 13.1 节所介绍。

软件多样性和冗余性可以被实现为多种不同的体系结构风格，本章的剩余部分将描述这些。

13.3.1 保护性系统

保护性系统是一种与其他系统相关联的专用系统。它通常是对一些过程的控制系统，比如化学制造过程或者一个设备控制系统，无人驾驶火车系统。保护性系统的例子可能是一个火车上探测火车是否正在通过一个红色信号灯的系統。如果是，并且没有迹象显示火车控制系统正在减速，那么保护性系统会要求系统制动使其停止。保护性系统独立地监控它的环境，并且如果传感器指示控制系统没有正确工作，那么保护性系统被激活并且关闭设备的运行。

图 13-3 说明了保护性系统与控制系统的关系。保护性系统监控控制设备及其环境。如果探测出来一个问题，它发出命令使执行器关闭系统或者触发其他保护机制（比如打开一个压力释放阀门）。注意：这里有两部传感器。一部用于正常的系统监控，而另一个专门供保护性系统使用。在传感器失效时，备用机制允许保护性系统继续操作。系统中可能还要有冗余的执行器。

保护性系统只包含要最紧要的功能，就是把系统从潜在的危险状态转换到安全状态（系统关闭）。它是一个更加通用的容错系统体系结构的一个实例，主系统由一个更小更简单的只包含关键功能的备用系统所支持。比如，美国航天飞机控制软件有一个备用系统包含“带你回家”功能；也就是，备用系统可以在主控制系统失败时让飞船安全着陆。

这种系统体系结构的优点是，保护性系统软件可以比控制被保护过程的软件简单得多。保护性系统的唯一功能是监视操作并确保紧急情况下系统被带回安全状态。因此，我们就可以投入更多的努力到容错和缺陷检测中去。你可以检查软件是否被正确描述并且是一致的，而且软件参照其描述也是正确的。目的就是确保保护性系统的可靠性，即在需要启动时其发生失败的可能是非常小的（如 0.001）。对保护性系统的调用应该是非常罕见的，对于请求失败的失败率为 1/1000，意味着保护性系统失败应该是非常稀少的。

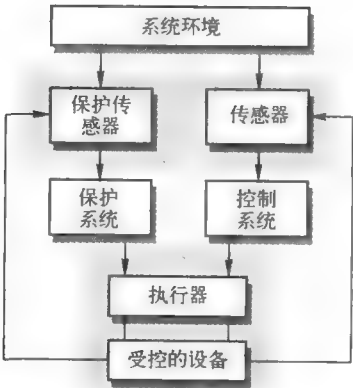


图 13-3 保护性系统的体系结构

13.3.2 自监控系统体系结构

自监控系统体系结构是指这样的一个系统体系结构，系统设计成监视其自身的操作并在问题探测到的时候采取某些行动。这是通过在不同的通道计算并比较计算的输出来实现的。如果输出是一致的并且同时可用，那么可以假设一个失败发生了。当其发生的时候，系统通常会在状态输出线上输出一个失败异常，这会使控制被转移到另一个系统上进行，如图 13-4 所示。

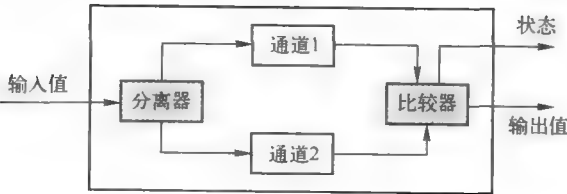


图 13-4 自监控系统体系结构

为了能有效地探测硬件和软件缺陷，自监控系统必须被设计成：

1. 在每一种通道上使用的硬件是多样的。现实中，这可能意味着每一种通道使用不同的处理器类型来实现要求的计算，或者是组成系统的芯片集可能源自不同的生产商。这降低了通常的处理器设计缺陷影响计算的可能性。

2. 在不同的通道上使用不同的软件。否则相同的软件错误可能会在同一时间在每一个通道上发生。我们将在 13.3.4 节讨论获得真正多样性软件的难度。

对其本身而言，此体系结构可以用于计算的正确性非常重要但可用性并不是至关重要的情况下。如果从不同的通道获得的答案是不同的，系统会简单地关闭。对于很多医疗和诊断系统，可靠性要比可用性更加重要，因为错误的系统反应会导致病人接受错误的治疗。然而，如果系统在错误事件中仅仅是关闭了，这虽然会带来不便但通常病人不会因此而受到系统的伤害。

在需要高可用性的情况下，你需要并行使用几个自监控系统。你需要一个切换单元来探测错误并在有两个通道产生一个一致的结果的时候从一个系统取得结果。这样的方法被用在空中客车 340 系列的飞行控制系统中，在这个系统中使用了 5 个自监控计算机。图 13-5 是一个简化的体系结构。

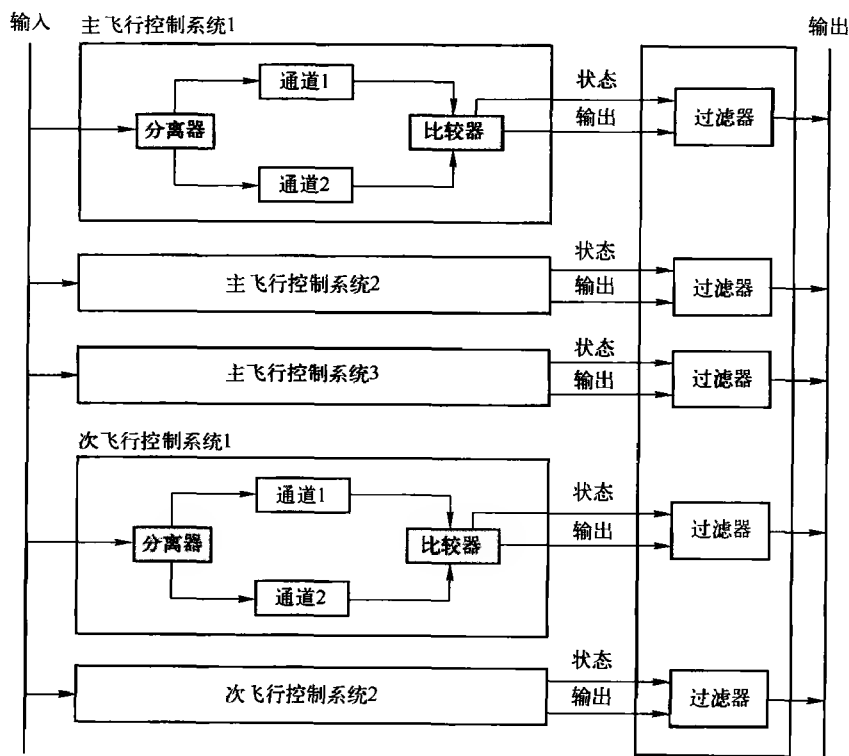


图 13-5 空中客车飞行控制系统体系结构

在空中客车飞行控制系统中，每一个飞行控制计算机并行地计算，使用的是同一个输入。输出被连接到一个硬件的过滤器上以检查是否这个状态指示的是一个缺陷。如果是，则这台计算机输出就被关闭。接下来从另一台替代计算机输出。因此，4 台计算机都失败而飞行操作依然继续是可能的。在 15 年多的运行当中，并没有报导过由于所有的飞行控制系统失败而导致飞行器失控的情况。

空中客车系统的设计者尝试过采用多种方法获得多样性：

- 1. 主飞行控制计算机使用与次飞行控制系统不同的处理器。
 - 2. 在主系统和次系统的每一个通道使用的芯片组来自不同的生产商。
 - 3. 次飞行控制系统中的软件只提供最紧要的功能——比主控制软件要简单。
 - 4. 主系统和次系统的每一个通道的软件使用不同的程序语言来开发，并且使用不同的开发团队。
 - 5. 在主系统和次系统中使用不同的编程语言。
- 如稍后要讨论的，这些都不能保证多样性，但是它们减少了不同通道共同失败的概率。

13.3.3 N - 版本编程

自监控系统体系结构是采用多版本编程提供软件冗余性和多样性的系统的例子。这个多版本编程的概念源自硬件系统中的三重组合冗余性（TMR）的概念。这个概念已经在构建容许硬件失败的系统中用了很多年（见图 13-6）。

在一个 TMR 系统中，硬件单元被复制 3 次（或者更多次）。每一个单元的输出被送到一个输出比较器上，这个比较器通常实现为一个投票系统。这个系统比较所有的输入，并且，如果两个或者更多的输入是相同的，那么其值就是输出。如果其中的一个单元失败了，并且产生的输出与其他单元产生的输出不一样，那么输出就被忽略。一个缺陷管理者可能自动尝试修改这个有缺陷的单元，但如果做不到的话，系统会自动地重新配置使那个发生缺陷的单元退出服务。系统会用剩余的两个单元继续工作下去。

这个容错方法依赖于大多数硬件失败的原因是组件失败造成的而不是设计缺陷造成的。这些组件是独立地失败的。它假设在正常情况下，所有的硬件单元都是按描述那样去工作的。因此所有的硬件单元同时发生组件失败的可能性是相当低的。

当然，这些组件可能都有一个普遍的缺陷并且因此都产生同样的（错误）答案。使用一个共同描述但由不同生产商设计和生产的硬件单元会降低这样的共同模式失败发生的几率。这一点所基于的假设是不同的团队做出相同的设计或生产错误的概率是很小的。

在容错软件中我们可以使用相似的方法，这就是 N 个不同版本的软件系统并行执行（Avizienis, 1985; Avizienis, 1995）。这个实现软件容错的方法在图 13-7 中说明，其被用于铁路信号系统、航空系统，以及反应堆保护性系统。

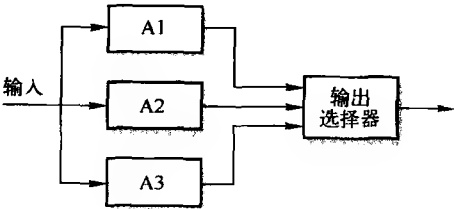


图 13-6 三重模块冗余

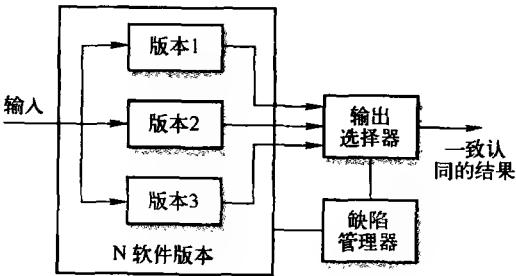


图 13-7 N - 版本编程

使用一个公用的描述，同一个软件系统由多个小组实现。这些版本在不同的计算机上执行。使用一个投票系统比较它们的输出，不一致的输出或者未能及时产生的输出被拒绝掉。至少系统的 3 个版本应该是可用的，所以在发生单个失败的情况下两个版本应该是一致的。

在需要高可用性的系统中，多版本编程可能比自监控体系结构要便宜一些。然而它仍然需要多个不同的小组开发多个软件版本。这导致很高的软件开发费用。结果，仅在无法为安全要求

极高的系统构造保护性系统的时候，才用到这个方法。

13.3.4 软件多样性

所有以上的容错体系结构都依赖于软件的多样性以获得容错功能。这一点基于的假设是，对于描述（对于保护性系统来说，或者是一部分描述）的不同实现之间是相互独立的。它们不应该包含同样的错误，这样也就不会以同样的方式同时失败。这需要软件由不同的小组来书写，而这些小组在开发过程中不应该交流，由此降低一致的误解或对描述的误读的几率。

采购系统的公司可能包含明确的多样性政策，以期最大化系统版本之间的差异。比如：

1. 需求中明确要求使用不同的设计方法。比如要求一个小组生产一个面向对象的设计而要求另一个小组生产一个面向功能的设计。
2. 规定程序要由不同的编程语言实现。比如，在3个不同的版本中，Ada、C++和Java被用来写不同的版本。
3. 要求对系统的开发要使用不同的工具和不同的开发环境。
4. 明确地要求在实现的某些部分用不同的算法。然而这限制了设计团队的自由，并可能因此难于与系统性能需求吻合。

每一个开发小组应该依据一个详细的系统描述（有时称做V-spec）工作，这个描述是从系统的需求描述中导出的（Avizienis, 1995）。这个描述应该被充分地细化以保证描述中没有二义性。在定义系统的功能的同时，详细的描述应该定义用来比较的系统输出应该在哪里生成。

理想情况，系统的多版本之间应该没有依赖性所以应该以不同的方式失败。如果是这样，多样的系统的总的可靠性可以通过每一个通道的可依赖性相乘获得。所以，如果每一个通道的请求失败概率是0.001，则一个三通道系统（通道之间独立）总的POFOD是比一个单通道系统的可靠性高100万倍。

然而现实当中，获得完全独立的通道是不可能的。实验证明独立的设计小组经常犯同样的错误或者对描述有同样的理解错误（Brilliant等, 1990；Knight和Leveson, 1986；Leveson, 1995）。有以下几个原因：

1. 不同小组的成员经常来自同一个文化背景，并可能在受教育时使用同样的教科书。这意味着他们可能觉得很难理解的事情是类似的，并且可能同样觉得与领域专家交流很困难。很可能他们独立犯下的错误和设计的算法都是一样的。
2. 如果需求是不正确的或是其基础是对系统环境的误解，那么这些错误将会反映到每一个系统的实现上。
3. 在要求极高的系统中，详细系统描述是一个基于系统需求的详细文档，对团队提供完整的细节，描述系统应该如何执行。软件的开发者是没有机会解释需求的。如果文档当中有错误，这些错误将会反映到所有的开发团队并且在所有的系统版本中实现。

一种有可能减少共同描述性错误发生的方法是开发详细的描述，即独立地开发详细描述，并使用不同语言进行描述。一个开发团队可能依据一个形式化描述来开发而另一个依据一个基于状态的系统模型来开发，而第三个依据自然语言描述来开发。这可以避免一些描述解释的错误，但无法逃避描述错误的问题。它也带来了在翻译需求的时候引入错误的机会，导致了不一致的描述。

通过对实验的分析，Hatton (1997)，总结得出：一个三通道大约比单通道系统的可依赖性高5~9倍。他认为，通过对单个版本投入更多资源以增进可依赖性是达不到这一点的，所以多版本方法比单版本方法可以更好地应用于可依赖系统。

然而不能确定的是，通过多版本增加可依赖性的额外开发费用是否值得。对于很多系统来

说，这样的额外花销可能是不合理的，因为一个设计良好的单通道系统或许已经足够用了。只有在安全和任务要求极高的系统中，由于失败的代价是高昂的，才真正需要多版本软件。即使在这样的情况下（比如航天系统），提供一个有限的功能简单的备份直到主系统能够被修复并且重启，可能也是足够的了。

13.4 可依赖的编程

总的来说，本书没有讨论编程问题，因为在没有涉及编程语言的具体细节的情况下讨论编程问题几乎是不可能的。现在有这么多的方法和语言应用在软件开发过程中，所以本书避免用某一种语言为例。然而，当考虑到可依赖性工程时，有一套已经被接受的好的编程实践，它们相当普遍而且可以减少在交付的系统中的缺陷。

图 13-8 显示了一个好的实践指导准则的列表。它们可以应用在任何一种系统开发使用的语言中，尽管它们所使用的方式取决于系统开发所使用的特定的语言和符号系统。

可依赖的编程实践指导准则：

1. 限制程序中信息的可见性；
2. 检查所有输入的有效性；
3. 为所有异常提供处理；
4. 最小化使用易于出错的结构；
5. 提供重启的能力；
6. 检查数组边界；
7. 当调用外部组件时加入超时处理功能；
8. 为每一个代表现实世界值的常量命名。

图 13-8 好的可依赖编程实践指导准则

指导准则 1：限制程序中信息的可见性

在军事机构中所采用的一个信息安全原则是所谓的“需要知道”原则。只有那些需要了解某条信息以便执行任务的人才给予此条信息。若信息不与他们的工作直接相关就不会给予他们。

在程序设计过程中，也要用类似的原则去控制访问系统所使用的变量和数据结构。程序组件应该只允许访问那些与自身实现相关的数据。其他程序数据应该是不可接近的，对组件式隐藏的。如果使用了信息隐藏，隐藏的信息就不会被无关组件所破坏。如果接口保持不变，数据表示的改变将不会影响到系统中的其他组件。

可以通过在你的程序中将数据结构定义为抽象数据类型来做到这一点。抽象数据类是一种数据形式，这种类型变量的内部结构和表示是隐藏的。类型的结构和属性从外部是不可见的，对数据的所有访问要通过操作实现。比如说，你可能用一个抽象数据类型来表示一个请求服务的队列。操作应该包括 get 和 put，也就是向队列里加入项或者从队列取出项，还要有获得队列中项数量的操作。可以在最初将这个队列实现为一个数组，但接下来又将其改变为一个链表。这可以在不改变针对队列的任何代码的前提下实现，因为队列的表示永远不会被直接访问。

还可以用抽象数据类型来检查赋值是否在合理范围之内。比如说，要表达化学反应过程中的温度，允许的温度范围是在 20 ~ 200 摄氏度之间。通过在抽象数据类型中包括一个对所赋的值的检查可以确保温度值永远不会超过要求的范围。

在一些面向对象的语言中，可以使用接口定义实现抽象数据类型，也就是声明一个对象的接口而不引用对对象的实现。比如说，可以定义一个接口 Queue，支持将对象加入队列，从队列取出对象，计算队列的大小等。在实现接口的对象类中，方法和属性都应该是

类私有的。

指导准则 2：检查所有输入的有效性

所有的程序都是从它们的环境中获取输入并处理它们。描述所做的假设是这些输入反映了它们在真实世界中的使用。比如，可以假设一个银行账户数字总是 8 位正整数。可是在很多情况下，系统描述没有定义在输入错误的情况下应该采取什么样的动作。人们不可避免地会犯错误，有时候会输入错误的数据。有时候，如本书在第 14 章将会讨论的那样，对系统的恶意攻击靠的就是错误的输入。甚至当输入是从传感器或者其他系统来的时候，这些系统也可能出错并提供不正确的值。

因此，每一次从程序的操作环境中输入数据的时候都应该验证输入的有效性。很明显这些检查要取决于输入本身，但以下这些可能的检查还是会用到的：

1. **范围检查** 可以认为输入应该在一个特定的范围之内。比如，一个代表概率的输入应该在范围 0 ~ 1.0 之间，代表液态水温度的输入应该在 0 ~ 100 摄氏度之间，等等。

2. **位数检查** 输入可以被看成是固定数量的字符的形式（比如 8 位字符表示的银行账户）。在其他情况，位数可能不确定但是可能有一个现实的最大上限。比如说，人名的字符数应该不会超过 40 个字符。

3. **表现检查** 输入应该是一个特定的样式，以一种标准的方式表示。比如说，人名不应该包括数字，电子邮件由两部分组成并以 @ 符号分开，等等。

4. **合理性检查** 输入是一系列输入中的一个，而你知道一些关于这个系列输入之间的关系，这样就可以检查输入值是否是合理的。比如说，如果输入值代表一个家庭用电电表的读数，那么你可以认为用电量应该大致与前一年同期用电量相仿。当然，变化是会有有的，但是巨大的变化往往意味着发生了问题。

当输入有效性验证检查失败的时候你所要采取的措施取决于所实现系统的类型。在某些情况下，你要将问题报告给用户并且要求重新输入这个值。如果一个值来自传感器，你可能使用最近的有效值。在嵌入式实时系统，你可能必须根据历史信息来估计值，这样系统才能继续执行。

指导准则 3：为所有的异常提供处理

在程序执行期间，错误或者难以意料的事件不可避免会发生。这些事情的发生或者是因为一个程序缺陷或者是不可预料的外部环境的结果。在程序执行期间错误或不可预料的事件被称作“异常”。异常的例子可以是系统电源失败，企图访问不存在的数据，或者是数值上溢或下溢。

异常可能是由硬件也可能是由软件条件决定的。当一个异常发生的时候，必须由系统来处理。这一点可以由程序本身完成或包含一个对系统异常处理的转换控制机制。一般来说，系统的异常管理机制报告错误并关闭执行。因此，为保证程序异常没有引起系统失败，应该为所有可能发生的异常定义一个异常处理，并保证所有的异常被探测出来并得到明确的处理。

在像 C 语言这样的编程语言中，必须用 if 语句来探测异常并将控制转移到异常处理代码。这意味着在程序中任何可能发生异常的地方都必须明确地检查异常。然而，这个方法明显增加了异常处理任务的复杂性，增加了犯错误的机会，并因此会对异常误处理。

有一些编程语言，比如 Java、C++ 和 Ada，包含支持异常处理的结构，所以可以不用额外条件语句来检查异常。这些程序语言包含一个特殊的内置类型（通常称为异常），不同的异常可以声明为这种类型。当一种异常情况发生时，产生异常信号而语言运行时系统将控制转移到异常处理器。这是一个声明异常名字并对每个异常提供相应的处理动作的代码段（见图 13-9）。注意异常处理器是在正常的控制流之外存在的，这个正常的控制流不会在异常处理

之后恢复执行的。

异常处理通常做以下 3 件事情中的一件或多件：

1. 向更高层组件发送信号报告一个异常已经发生，并提供异常类型的信息。当一个组件调用另一个组件的时候使用这个方法，发出信号的组件必须知道被调用的组件是不是成功执行了。如果没有，要由调用组件采取动作来从这个问题中恢复。

2. 实现一些代替处理来替换原始准备的处理。这样，异常处理器要采取一些措施来从问题中恢复。处理过程能如平常一样继续或者异常处理器能指示一个异常已经发生以提醒调用组件问题的存在。

3. 传递控制到能处理异常的运行时支持系统。当程序发生错误的时候（比如说当发生一个数值溢出），这经常是默认的处理。

通常运行时系统采取的动作是暂停处理过程。在将控制转给运行时系统之前，如果可能将系统转移至一个安全和静止的状态时，这是唯一应该使用的方法。

在程序中处理异常使得检测和恢复一些输入错误以及未预料到的外部事件成为可能。从某种意义上讲，它提供了一种容错层次——程序探测到错误并且能够采取动作从这些错误中恢复。因为大多数的输入错误和未预料的外部事件通常都是瞬态的，所以往往在异常得以处理之后可以继续正常运行。

指导准则 4：最小化使用容易出错的结构

程序中有缺陷，进而造成很多程序失败，是源于人所犯的错误。程序员因未能很好跟踪状态变量之间的很多关系而犯错误。他们写的程序语句导致未预料到的行为和系统状态改变。人们总是会犯错，但在 20 世纪 60 年代末期，人们逐渐认识到了有些编程方法比其他编程方法更容易向程序中引入错误。

一些编程语言的结构和编程技术本质上容易犯错，因此应予以避免，或者至少尽量少用。潜在的容易犯错误的结构有：

1. **无条件分支（go-to）语句** go-to 语句的危险早在 1968 年就被发现（Dijkstra, 1968），这种语句因此被现代编程语言所弃用。然而，它们仍然在像 C 语言这样的语言中使用。go-to 语句的使用导致“意大利面条代码”的产生，非常混乱并难以理解和调试。

2. **浮点数** 浮点数在定长字中的表示具有固有的不准确性。在对它们做比较操作时会出现一个独特的问题，因为表示上的不精确可能造成无效的比较。举例来说，3.000 000 00 有时可能表示成 2.999 999 99，也可能表示成 3.000 000 01。比较会得出两个不相等的结论。定点数用确定的小数位数来表示一个数要安全得多，因为做精确的比较是可能的。

3. **指针** 编程语言比如 C 或 C++ 支持称为指针的低级结构，它保存一个直接指向机器内存（它们指向内存位置的指针）。如果指针赋值错误并因此指向了错误的内存区域，那么指针使用的错误可以是灾难性的。对指针来说，做数组和其他结构的边界检查是难以实现的。

4. **动态内存分配** 程序内存可以是在程序运行时分配的而不是编译时分配的。这种做法的危险是内存不能被正确释放，因此，程序最终可能将内存耗光。这可能是一种非常难以检测的错误类型，因为在问题发生之前，这个系统可以成功地运行很长一段时间。

5. **并行** 当过程同时被执行，它们之间可能存在某种微妙的时序关系。时序问题通常没法通过程序检查搞清楚，而且一些容易引起时序问题的奇特的组合情形不会在系统测试中发生。并行虽然是不可避免的，但是可以对它进行仔细地控制来减少过程间的依赖性。

6. **递归** 递归是指这样的一种情形，一个程序调用它本身或者调用另外的一个程序，而这

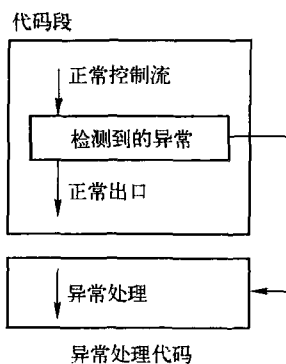


图 13-9 异常处理

个被调用的程序又调用了最初调用它的那个程序，这就是“递归”。递归的使用能使程序简洁，但是跟踪递归程序的逻辑是很困难的。所以程序设计中的错误是很难检测出来的。在临时堆栈变量创建的时候，递归错误会导致系统所有内存都分配完毕。

7. 中断 中断是一种手段，用来将控制强制性地转移到与目前程序执行不相关的一段代码上。中断的危险是显而易见的，它会导致一些关键操作被迫终止。

8. 继承 面向对象程序语言中继承的问题是，与对象关联的代码并不总在一处，这使得对对象行为的理解更加困难。因此，程序设计错误将更容易被忽略。此外，当继承与动态绑定在一起使用时可能造成运行时的时序问题。同一方法的不同实例可以被绑定到一起调用，这取决于参数类型。结果是，寻找正确的方法实例将会花费不同的时间。

9. 别名 这是指使用不同的名字来指向程序中的同一个实体；比如说，如果两个相同名称的指针指向同一个内存地址。当有多个名字与一个对象实体相连时，程序读者很容易忽略对实体修改的语句。

10. 无界数组 在一些语言中，如C中，数组是访问内存的简单方法，你可能赋值到数组的边界之外。运行时系统并不检查赋值真正是对数组内元素的赋值。数组越界，即在有攻击者故意编程往数组的那一段缓冲区之外的内存写值时，就形成了信息安全的一个脆弱性。

11. 默认输入的处理 一些系统提供默认的输入处理，不管什么值输入到系统中。这是一个信息安全漏洞，攻击者可以利用它输入一个不会被程序拒绝的输入。

有些安全性要求极高的系统的开发标准完全禁止这些结构的使用。不过，这种极端做法有时也是不实际的。这些结构和技术是很有用的，但必须小心使用。只要有可能，就应该在抽象数据类型或对象中使用，从而控制潜在的危险。如果错误发生，抽象数据类型和对象就担当了类似“防火墙”的角色，限制了损失的程度。

指导准则5：提供重启能力

很多机构信息系统都是基于短事务的，处理用户的输入需要较短的时间。这些系统被设计成仅当所有的过程都成功完成之后才做出对系统数据库的改动。如果在处理的过程中有错误，数据库将不会被更新，因此不会出现不一致的情况。事实上，所有的电子商务系统，只要是你最后过程确认你的购买的系统，都是以这种方式工作的。

用户与电子商务系统的交互总是持续几分钟时间并包含最少的处理。数据库事务很短，并且通常可以在少于一秒钟之内完成。然而其他种类的系统比如CAD系统和文字处理系统的处理时间比较长。在长处理系统中，开始使用系统到结束任务可能要花费几分钟的时间甚至几小时。如果系统在长时间处理失败，那么所有的工作可能都会丢失。相似地，在计算紧密型系统中，比如一些科学计算系统，可能需要几分钟甚至几小时的处理才能完成计算。如果发生系统失败，那么所有的时间将损失掉。

在所有这类系统中，应该提供一个重启能力，目的是保持在处理过程中收集或生成的数据的副本。重启功能应该允许系统使用这些副本重新开始，而不是从头开始。这些副本有时被称为检查点。比如：

1. 在电子商务系统中，可以保存用户所填的表单并允许他们访问和提交这些表单而不用让他们再次填写。

2. 在长处理或计算密集型系统中，可以每几分钟自动保存数据并且在系统失败的时候可以用最近保存的数据重新开始。同时，应该允许用户错误，提供一种用户能够返回到最近的检查点并从那里重新开始。

如果异常发生而且正常的操作无法再继续下去，可以使用向后错误恢复来处理异常。这意味着要将系统状态重置为检查点处保存的状态并从这个点重启操作。

指导准则 6：检查数组边界

所有的程序语言都允许数组描述，即使用一个数字索引访问一个线性数据结构。这些数组通常设置在程序的工作内存的一个连续的区域。数组被描述成一种特定的长度，其长度反映出它们是如何被使用的。比如说，如果你要表示不超过 10 000 个人的年龄，那么你可能会声明一个包含 10 000 个存储年龄数据的数组。

一些编程语言，比如 Java，在每次对数组输入值的时候都做检查，检查索引是否在数组当中。所以如果一个数组 A 的索引是从 0 到 10 000，试图访问元素 A[-5] 和 A[12345] 会导致异常的发生。然而编程语言如 C 和 C++ 不会自动地做数组边界检查的工作，而仅仅简单地计算一个从数组的开始位置算起的偏移量。因此，A[12345] 会访问从数组开始位置计算的 12345 个位置的字节，不管这个字节是不是数组的一部分。

这些语言不做自动数组边界检查的原因是这会在每次数组访问的时候引入额外开支。大部分数组访问是正确的，所以边界检查多数情况下是不必要的，并且会增加程序的执行时间。然而缺少边界检查会导致安全性的缺乏，比如第 14 章将介绍的缓冲溢出。更常见的是，它会向系统引入脆弱性而可能导致系统失败。如果你在使用一个没有数组边界检查的语言，你总是要增加额外代码来保证数组的索引是在边界之内。如指导准则 1 中所说的，这一点可以通过将数组实现为抽象数据类型轻松达到。

指导准则 7：当调用外部组件时加入超时处理功能

在分布式系统中，系统的组件在不同的计算机上执行，调用要在网络上从一个组件传送到另一个组件。为获得一些服务，组件 A 可以调用组件 B。A 在继续执行之前等待 B 的响应。但是如果组件 B 由于某些原因没能做出响应，那么组件 A 就不能继续执行。它为一个响应无限等待下去。等待系统响应的人看到的是一个沉默的系统失败，没有系统的响应。没有别的办法只能结束等待进程并重新开始系统。

为了避免这样，调用外部组件的时候应该总是包含超时机制。超时是自动地假设一次组件调用已经失败并不会产生响应。你要定义期待从一个被调用组件接收到响应的时间段。如果没有在这个时间段里接收到响应，可以认为调用已经失败并从被调用组件那里收回调用。你可以接下来尝试从失败中恢复或告诉系统用户发生了什么事情和允许他们做什么。

指导准则 8：为每一个代表现实世界值的常量命名

所有不是太简单的程序都包含一定数量的常量值来表示真实世界实体的值。这些值不会随着程序的执行而改变。有时候，它们是从不改变的常量（比如说光的速度），但是更多的情况下它们是随时间相对缓慢地改变的值。比如说，一个计算个人所得税的程序将会包含表示当前税率的常量。这些常量将会年复一年地变化，所以程序必须用新的常量值更新。

你应该在你的程序中加入一个片段用来命名所有你用到的现实世界常量值。当使用常量的时候，你应该用名字来指代它们而不是用值。考虑到可依赖性这样做有两点好处：

1. 可以少犯错误或少使用错误的值。我们很容易打错一个数字，系统往往不能发现错误。比如税率是 34%。一个很简单的换位错误可能导致将其误打成 43%。但是如果你错打了一个名字（如标准税率），编译器通常因未声明该变量而将其探测出来。
2. 当一个值改变的时候，你不用查找整个程序来发现在哪里使用过这个值。你所需要做的仅仅是改变与这个值相关的常量声明。新的值会自动在任何需要的地方出现。

要点

- 程序的可依赖性能通过以下方式达到，即避免导入缺陷，在系统部署之前检查并清除缺陷以及包括容错设施，容错设施容许系统在由于缺陷引发了失败的情况下仍然能继续工作。

- 软件过程和软件系统中的冗余性和多样性的利用对可依赖系统的开发是必须的。
- 一个良好定义的、可重复使用的过程对于最小化系统缺陷是十分重要的。在从需求定义到系统实现的所有阶段中这个过程应该都包括检验和有效性验证活动。
- 可依赖系统体系结构是为容错设计的系统体系结构。有很多种体系结构风格支持容错，包括保护性系统、自监控系统体系结构和多版本编程。
- 实现软件多样性很难，因为现实中不能保证软件的每个版本都是真正独立的。
- 可依赖编程利用程序中包含冗余来检查输入的有效性和程序变量的值。
- 一些程序设计结构和技术，例如 go-to 语句、指针、递归、继承以及浮点数等，是天生易发生缺陷的。在开发可依赖的系统时不应该使用这些内容。

进一步阅读材料

《Software Fault Tolerance Techniques and Implementation》一个关于实现容错和容错体系结构技术的综合性论述。这本书也包括关于软件的可依赖性的一般问题（L. L. Pullum, 2001, Atrech House）。

《Software Reliability Engineering: A Roadmap》这篇论文是由一个软件可依赖性的前沿专家写的。概述了软件可依赖性工程的技术发展现状并讨论了研究的挑战（M. R. Lyu, Proc. Future of Software Engineering, IEEE Computer Society, 2007）。<http://dx.doi.org/10.1109/FOSE.2007.24>。

练习

- 13.1 为什么公司想确保他们的软件无错是不划算的？请给出 4 个原因。
- 13.2 为什么认为使用可依赖性过程进行可依赖软件的创建是合理的。
- 13.3 给出在可依赖性过程中可能结合的两个不同的、冗余的活动的例子。
- 13.4 什么是支持软件容错的体系结构风格的共有特性。
- 13.5 想象你在实现一个基于软件的控制系統。提出一种环境使其适合使用一个容错结构，解释为什么需要这种方法。
- 13.6 你负责设计一个通信交换机，必须提供 24/7 的可用性，但不是安全性要求极高的。给出你答案的理由，并建议一种可以用在这个系统的体系结构风格。
- 13.7 有人建议在治疗癌症的射线理疗仪的控制软件开发中使用 N - 版本程序设计。你认为这是否是一个好的提议并说明原因。
- 13.8 给出两个理由说明基于软件多样性的不同的系统版本都会经历同样的失败。
- 13.9 为什么应该在可用性要求极高的系统中，要对所有的异常进行明确处理。
- 13.10 使用本章中讨论的技术来开发安全的软件显然要增加可观的额外成本。如果在系统的生存期的 15 年中挽救了 100 个人的生命，什么样的额外成本看来是可以接受的？如果只有 10 个生命得到挽救，上面的成本还是可以接受吗？人的收入能力会影响成本分析吗？

参考书目

- Avizienis, A. (1985). 'The N-Version Approach to Fault-Tolerant Software'. *IEEE Trans. on Software Eng.*, SE-11 (12), 1491-501.
- Avizienis, A. A. (1995). 'A Methodology of N-Version Programming'. In *Software Fault Tolerance*. Lyu, M. R. (ed.). Chichester: John Wiley & Sons. 23-46.
- Boehm, B. (2002). 'Get Ready for Agile Methods, With Care'. *IEEE Computer*, 35 (1), 64-9.
- Brilliant, S. S., Knight, J. C. and Leveson, N. G. (1990). 'Analysis of Faults in an N-Version Software

Experiment'. *IEEE Trans. On Software Engineering*, **16** (2), 238–47.

Dijkstra, E. W. (1968). 'Goto statement considered harmful'. *Comm. ACM.*, **11** (3), 147–8.

Hatton, L. (1997). 'N-version design versus one good version'. *IEEE Software*, **14** (6), 71–6.

Knight, J. C. and Leveson, N. G. (1986). 'An experimental evaluation of the assumption of independence in multi-version programming'. *IEEE Trans. on Software Engineering.*, **SE-12** (1), 96–109.

Leveson, N. G. (1995). *Safeware: System Safety and Computers*. Reading, Mass.: Addison-Wesley.

Lindvall, M., Muthig, D., Dagnino, A., Wallin, C., Stupperich, M., Kiefer, D., May, J. and Kahkonen, T. (2004). 'Agile Software Development in Large Organizations'. *IEEE Computer*, **37** (12), 26–34.

Parnas, D. L., Van Schouwen, J. and Shu, P. K. (1990). 'Evaluation of Safety-Critical Software'. *Comm. ACM*, **33** (6), 636–51.

Pullum, L. L. (2001). *Software Fault Tolerance Techniques and Implementation*. Norwood, Mass.: Artech House.

Storey, N. (1996). *Safety-Critical Computer Systems*. Harlow, UK: Addison-Wesley.

Torres-Pomales, W. (2000). 'Software Fault Tolerance: A Tutorial.'

http://ntrs.nasa.gov/archive/nasa/casi./20000120144_2000175863.pdf.

信息安全工程

目标

本章的目标是介绍有关安全的应用系统设计中所需要考虑的诸多问题。读完本章，你将了解到以下内容：

- 了解应用级信息安全和基础设施级信息安全之间的区别；
- 理解如何使用生命周期的风险评估和运行的风险评估来理解影响系统设计的信息安全性问题。
- 了解信息安全软件开发的软件体系结构和设计准则。
- 了解系统生存力的概念以及为什么对于复杂软件系统来说生存力分析是重要的。

20 世纪 90 年代因特网的广泛使用给软件工程师带来了新的挑战，即需要设计和实现能保证信息安全的系统。随着越来越多的系统与因特网的连接，系统面临着各种各样的外部攻击，这些人为设计的攻击严重威胁着系统的安全。生产可依赖系统的问题在急剧增加。系统工程师不得不考虑来自怀有恶意的和技术能手这两类攻击者的威胁，也要考虑开发过程中一些意外人为错误所导致的众多问题。

现在看来设计能抵御外部攻击并在遭受攻击后成功恢复的系统是十分必要的。没有对安全的预防，网络上的系统不可避免地受到攻击者的危害。攻击者可能滥用系统硬件、盗窃机密数据或者是破坏系统所提供的正常服务。因此，系统的信息安全工程正逐渐变成系统工程过程的一个重要方面。

信息安全工程关系到系统开发和进化的过程，这样的系统要能抵御恶意的攻击，保护系统以及它的数据。信息安全工程是计算机安全的一个更为广泛的领域。它已经成为商业和个人的首要问题，因为越来越多的罪犯试图通过网络系统从事犯罪活动。软件工程师应该意识到系统所面对的信息安全的严峻局面，并有能力去迎接这种挑战。

本章的目的是向软件工程师介绍信息安全工程，着重介绍影响应用级信息安全的设计问题。这一章不是对计算机信息安全做全面的概述，所以不会涉及密码、访问控制、授权机制、病毒以及木马等话题。这些内容在计算机信息安全方面的专门教科书中有更为全面的讲述（Anderson, 2008；Bishop, 2005；Pfleeger 和 Pfleeger, 2007）。

本章增加了其他章节中有关安全话题的一些讨论。读者可以与下面的关联内容一起阅读此部分：

- 10.1 节，介绍了信息安全性和可依赖性是怎样紧密联系在一起的；
- 10.4 节，介绍了信息安全性术语；
- 12.1 节，介绍了风险驱动描述的概念；
- 12.4 节，介绍了信息安全性需求描述的一般问题；
- 15.3 节，介绍了一系列信息安全性测试的方法。

在我们讨论信息安全这个话题的时候，我们需要考虑应用软件（控制系统、信息系统等）和支撑此类系统的基础设施两个方面（见图 14-1）。对于复杂应用来说其基础设施包括：

- 操作系统平台，例如 Linux 或者是各种 Windows 操作系统；

- 系统上运行的其他的通用应用，如 Web 浏览器和电子邮件客户端程序；
- 数据库管理系统；
- 支持分布式计算和数据库访问的中间件；
- 用于应用软件开发的可复用组件库。

应用
可复用组件和库
中间件
数据库管理
通用、共享的应用（浏览器、电子邮件等）
操作系统

图 14-1 存在信息安全性漏洞的系统分层

绝大多数的攻击都是集中在对系统基础设施的攻击上，这是因为这些基础设施组件（例如 Web 浏览器）是公开的和非常容易得到的。攻击者能够探测到这些系统的漏洞，并且共享他们发现系统漏洞的信息。正是由于很多人都使用同一种软件，攻击才具有了广泛的适用性。基础设施级的漏洞可能让攻击者获得对应用系统和数据的未授权的访问。

在实践中，应用级信息安全和基础设施级信息安全是有很大区别的：

1. 应用级信息安全是个软件工程问题，软件工程师应该确保系统是按照抵御攻击的要求去设计的。
2. 基础设施级信息安全是个管理问题，系统管理者对基础设施进行配置以抵御攻击。系统管理者需要设置基础设施，使得绝大多数对任何基础设施的信息安全特性的有效使用都能够很好地满足。他们也需要修补基础设施在信息安全上的漏洞，以便应用软件不会受到这些漏洞的影响。

信息安全管理不是一个孤立的任务，它包括一系列活动，如用户和权限管理，系统软件部署和维护以及攻击监控、测试和恢复：

1. 用户和权限管理包括向系统添加和删除用户，保证适当的用户认证机制的有效，设置系统的权限让用户只能访问到他们应该访问的那些资源。
2. 系统软件部署和维护包括系统软件和中间件的安装，对这些系统的正确配置保证安全漏洞不会出现。它也包括不断地用新版本或补丁更新系统，及时修补发现的安全问题。
3. 攻击监控、检测和恢复包括的活动有：对未授权的系统访问的监视、检测和及时有效的抵御攻击和备份活动，以保证常规性的操作能在遭受外部攻击后得以恢复。



内部攻击和社会工程

内部攻击是指那些由可信赖的个体（内部人员）滥用这种权利作用于系统上的攻击。例如，在医院里工作的护士，可以获得自己并不关心的但是属于病人个人隐私的医疗记录。应对内部攻击是有困难的，因为要使用附加的安全技术，这样可能扰乱那些可靠的系统用户。

社会工程是一种欺骗那些获得认可的用户开放他们凭证的方式。攻击者因此可以像一个内部人员那样进入系统。

<http://www.SoftwareEngineering-9.com/Web/SecurityEng/insiders.html>

信息安全管理是至关重要的，但是通常人们不认为是应用级信息安全工程的一部分。实际

上,应用级信息安全工程关注于所设计的系统在给定的预算和可用性约束条件下尽可能的安全。部分过程是“为管理的设计”,我们设计系统使得因信息安全管理错误导致系统受到攻击的可能性降到最小化。

对于要求极高的控制系统和嵌入式系统,通常的做法是选择一个合适的基础设施来支持应用系统。例如,嵌入式系统开发者通常选择一个实时的操作系统来提供给嵌入式应用它需要的设施。已知的脆弱性和信息安全需求会被考虑在内。这就意味着在信息安全工程中会采用整体的方法。应用级信息安全需求能够通过基础设施或者应用自身来实现。

然而,机构中的应用系统通常使用已存在的基础设施(操作系统、数据库等)来实现。因此,使用基础设施和它的信息安全特性的风险必须考虑在内,作为系统设计过程的一部分。

14.1 信息安全风险管理

对于有效的信息安全工程来说信息安全的评估和管理是非常必要的。风险管理主要是评估系统资产在攻击中的可能损失,并在这些损失和降低这些损失的成本之间选择一个平衡。信用卡公司一直在做这件事。通过引入一种新技术来降低信用卡欺诈是一种相对比较简单的办法。但是,通常来说直接补偿用户由于欺骗而遭受的损失要比购买和部署新的防欺骗技术成本小很多。在成本下降和攻击增加的情况下,这种平衡会发生改变。例如,信用卡公司现在开始用卡上芯片来代替磁条了。这样卡的拷贝就变得更加困难了。

风险管理与其说是技术问题不如说是个管理问题,所以软件工程师应该决定哪些控制应该包含在系统内。高级管理者要决定是接受信息安全的代价还是接受缺乏安全流程的后果。相反,软件工程师的任务是在信息安全问题上提供有用的技术指导和判断。因此,他们在风险管理过程中是十分重要的参与者。

正如第12章中所提到的,对风险评估和管理过程来说,关键是机构的信息安全策略。机构的信息安全策略是要贯彻到所有的系统中的,并应该明确什么是应该做的什么是不允许发生的。信息安全策略要给出关于环境的说明,此环境是信息安全系统要一直维护的环境,以帮助判断可能出现的威胁。因此,信息安全策略明确了哪些是被允许的,哪些是不允许的。在信息安全工程过程中,我们设计机制来实现这种策略。

风险评估要在做出系统决策之前开始,并应该贯穿整个系统开发过程和系统投入使用之后(Alberts和Dorofee,2002)。在第12章中同样有介绍,风险评估是一个阶段性的过程:

1. **初步风险评估** 在此阶段,尚未对具体的系统需求、系统设计或实现技术做出决定。此评估过程的目的是判定是否一个适当的信息安全等级能在合理的成本范围内达到。如果可以,我们就能够导出要实现的系统的特殊的信息安全需求。我们没有系统潜在漏洞的有关知识,也没有可复用系统组件或中间件的控制方面的知识。

2. **生存期风险评估** 此风险评估发生在系统开发整个过程中,可以借助系统设计和实现的技术决策信息进行。评估的结果可能会改变信息安全性的需求和附加新的需求。识别出已知和潜在的漏洞,并用于系统如何实现、测试和部署的决策。

3. **运行风险评估** 在系统部署和投入使用之后,风险评估应该继续注意系统是怎样使用的以及新的或是变更的需求的请求。当系统定义时对系统运行需求做出的假设可能是不正确的。机构的改变可能意味着系统在以不同于原计划的方式使用。因此,运行风险评估会产生新的信息安全需求,这些需求必须在系统进化升级时被实现。

初步风险评估的目标是导出整个系统的信息安全需求。在第12章中,会讲到一系列初始的信息安全需求怎样从一个初步风险评估中导出。本节将主要集中讲生存期风险评估和运行风险评估,由此来说明系统描述和设计是怎样受到系统所采用的技术的影响和系统使用方式

的影响。

进行风险性评估，我们需要明确对系统有可能产生的威胁。一种方法是，开发一个“误用案例”集（Alexander, 2003；Sindre 和 Opdahl, 2005）。前面已经讨论过用例（和系统的典型交互）在导出系统需求时是如何使用的。误用案例是表现与系统进行恶意交互的脚本。我们可以使用这些用例来讨论和明确可能的威胁，因此也就能够确定系统的信息安全需求。当要导出系统需求时，这些案例同样可以和用例结合在一起使用。

Pfleeger 和 Pfleeger（2007）描述了 4 个方面的威胁，在识别可能的误用案例时，可以作为出发点。这 4 方面如下：

1. 拦截威胁，该威胁允许攻击者获得对某些资产的访问。因此，对于 MHC-PMS 一个可能的误用案例可能是攻击者获得了某个名人的就诊记录。

2. 中断威胁，该威胁允许攻击者使得系统的部分不可用。所以，一个可能的误用案例可能是一次对于系统数据库服务器的拒绝服务攻击。

3. 修改威胁，该威胁允许攻击者篡改系统资产。在 MHC-PHS 中，这可能就是一个攻击者更改病人记录信息的误用案例。

4. 伪造威胁，该威胁允许攻击者向一个系统中插入错误信息。这在 MHC-PMS 中可能算不上是一个严重的威胁，但是对于银行系统来说绝对是个威胁，将钱转入犯罪者的账户这样的虚假交易可能被加入系统。

误用案例不仅仅在初步风险中 useful，在生存期风险分析和运行风险分析的信息安全分析中也会被使用到。这些用例对于针对系统的所有的假设攻击和对于所做出的设计决定的信息安全影响的评估提供了有用的根据。

14.1.1 生存期风险评估

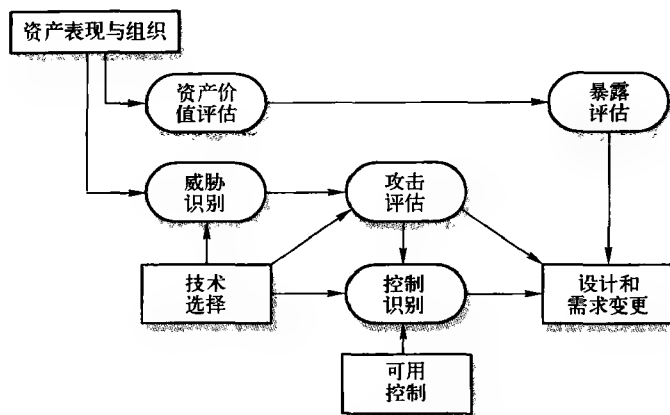
基于机构的信息安全策略，初步风险评估应当明确系统最重要的信息安全需求。这些需求反映了信息安全性策略应当怎样在应用中实现，明确被保护的资产，以及决定使用什么样的方式提供保护。但是，维护信息安全是要关注细节的。要求初始的信息安全需求考虑到所有可能影响安全的细节是不太可能的。

生存期风险评估明确影响信息安全的设计和实现的细节，这是生存期风险评估和初步风险评估之间的重要区别。生存期风险评估影响对已存在的信息安全需求的解释，产生新的需求，并且影响系统的整个设计。

在这个阶段评估风险时，我们会有更多关于需要保护哪些部分的详细信息，我们也会摸清系统中漏洞的有关情况。其中有些漏洞对于设计选择决断来说总是固有的。例如，对于基于口令的系统来说其漏洞是授权用户会将口令泄漏给未经授权的用户。或者，如果一个机构有一个用 C 语言开发软件的策略，我们会知道应用可能存在缺陷，因为这种语言并不包含数组边界检查。

信息安全的风险评估应该是系统的从需求工程到系统部署全生存期活动中的一部分。其过程类似于初步风险评估过程，只是增加有关漏洞识别和评估活动。风险评估的成果是一系列工程决断，它们影响系统设计或实现或者是限制其使用范围和方式。

图 14-2 表示了一个生存期风险分析过程的模型，它基于在图 12-9 中描述的初步风险分析过程。最重要的区别是，现在我们拥有了关于信息表示和分布，以及高级别的受保护资产的数据库组织方面的信息。我们同样知道了像软件复用、基础设施控制和保护等这些重要的设计决策。基于这些信息，我们的分析明确了信息安全性需求和对重要系统资产提供附加保护的系统设计所要做出的变动。



两个例子说明了保护需求是怎样受到信息表示和分布决策影响的：

1. 我们可能会做出一个设计决策来将私人患者信息和其接受治疗的信息分离开，通过一个键值（key）链接这些记录。治疗信息远不如患者个人信息敏感，所以不需要大量保护。如果键值被保护，那么攻击者就只能获得常规信息，而不能链接到患者个人信息。

2. 假设，在本节开始，设计决策是将患者记录拷贝到本地客户端系统。在服务器出现故障时也将允许工作继续进行。这让卫生保健的工作者，即使在没有网络连接的情况下也能够从笔记本电脑中获取患者记录。但是，我们现在有了两套要保护的记录和客户端副本，它们容易面临额外的风险，比如，盗取笔记本电脑。因此，我们必须思考什么样的控制应该用来降低风险。例如，笔记本电脑上的客户端记录应该被加密。

为了说明开发技术上的决定如何影响信息安全性，让我们假设医疗部门已经决定了用商业现货信息系统建立一个 MHC-PMS（他们自己的心理疾病病人信息管理系统）来保存病人的记录。此系统需要为使用它的每一类诊所做不同的配置。此决定已经做出，因为它会给绝大多数普遍使用的功能带来最低的开发成本和最快的开发时间。

当你基于某个现存系统建立应用，就必须接受系统原开发者的设计决策。让我们假设下列设计决策：

1. 系统用户是通过登录用户名/口令组合被鉴别（认证）的。没有其他鉴别方法可以使用。
2. 系统的体系结构是客户机 - 服务器，客户通过客户端个人计算机上标准的 Web 浏览器访问数据。
3. 信息以可编辑 Web 表格形式展现在用户面前。他们可以适当地改变信息并上传给服务器。

对于一个通用系统，这些设计决定是完全可以接受的，但是生存期风险分析表明它们都有相关的脆弱性。一些可能的脆弱性的例子如图 14-3 所示。

漏洞一旦被发现，我们就必须决定采取哪些措施来降低相应的风险。这一般包括是否添加额外的系统安全需求或者是其他的系统使用的操作过程。这里无法讨论所有的有关固有漏洞的需求，下面列出其中一些需求：

1. 口令检查程序应该是可用的且应该每天运行。登记在系统字典中的用户口令会被识别并报告给系统管理员。
2. 对系统的访问只能是在由系统管理员所认可和注册了的客户计算机上进行。
3. 所有的客户计算机都只能安装一个由系统管理员所认可的 Web 浏览器。

在使用商业现货系统时，系统自身不可能包含一个口令检查程序，所以必须使用一个独立

的系统。口令检查程序在用户密码被设定时分析其强度，如果用户选择了一个强度很弱的密码，那么它会通知用户。因此，易受攻击的口令能被很快发现并采取措施让用户更改其口令。

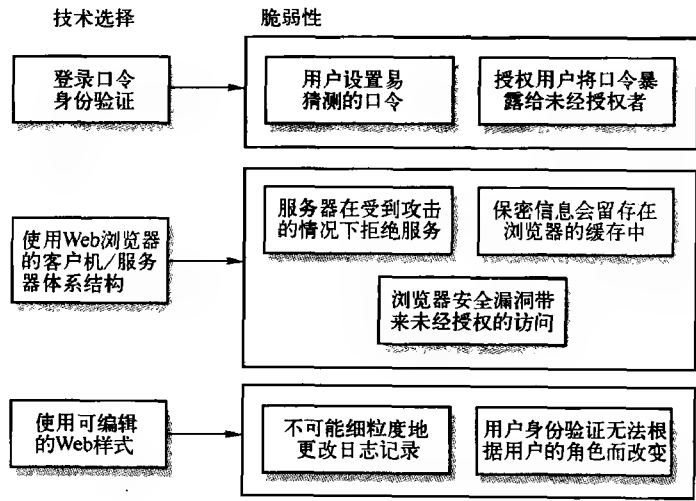


图 14-3 与技术选择相关的脆弱性

第二个和第三个需求意味着用户将总是从同一个浏览器上访问系统。我们可以在系统部署的时候认定哪个浏览器是最安全的，并且安装它到所有客户端的电脑。信息安全更新得到简化，因为没有必要在发现和修补信息安全漏洞的时候更新不同的浏览器。

14.1.2 运行风险评估

信息安全风险评估应该贯穿系统的全生命周期，来识别出现的风险和可能需要应对这些风险的系统变更。这个过程叫做运行风险评估。因为系统需求的改变，系统基础设施的改变，或是系统使用环境的改变，新风险可能出现。

运行风险评估过程和生命期风险评估过程类似，但是有更多的关于系统运行环境的信息。因为环境的特性能够导致系统产生新风险，所以环境很重要。例如，假设系统的用户会在使用时频繁遭到打断。一个风险就是这种打断意味着用户离开电脑，使其在无人管理的状态。对于一个未经授权的人来说，这样就可能获得系统中的信息。这可能会产生一个口令保护屏保的需求，口令屏保在短期的不活动状态之后启动。

14.2 面向信息安全的设计

一般来说，在一个系统已经实现后，要想将信息安全性再加入系统是很困难的。因此，我们需要在系统设计过程中就将信息安全问题考虑在内。在这一部分中，主要关注系统设计问题，因为其他的信息安全书籍中都没有对这个话题给予应有的重视。实现问题和人为错误会对信息安全产生重要影响，但是这些通常与所使用的特殊技术有关。推荐 Viega 和 McGraw 的书（2002），这是一本信息安全编程的导论。

本节集中讨论如下几个一般性的且应用独立的安全系统设计问题：

- 1. 体系结构设计，体系结构设计决策是如何影响系统信息安全的？
- 2. 好的实践，什么是在设计安全系统时的公认的好的实践？
- 3. 部署的设计，在系统使用前的部署时，为避免导入漏洞应该为系统设计什么样的支持？



拒绝服务攻击

拒绝服务攻击试图通过大量的服务请求去轰击 (bombarding) 一个网络系统, 以此来造成这个网络系统的瘫痪。这样的系统负荷, 因为没有在设计之初考虑, 所以它们会排除合法的请求。因此, 系统会因为过重的负荷而崩溃, 或者系统管理员将其脱机以停止大量请求的涌入, 这样都会导致系统变得不可用。

<http://www.SoftwareEngineering-9.com/Web/Security/DoS.html>

当然, 这些还不是仅有的对信息安全重要的设计问题。每一个应用都是不同的, 信息安全设计也必须考虑到应用的用途、临界状态和操作环境。例如, 如果开发一个军事系统, 我们需要采用他们的信息安全分类模型 (秘密、机密、绝密等)。如果要设计一个存储个人信息的系统, 我们要考虑到数据保护法案, 该法案是对信息如何管理施加的限制。

在可靠性和信息安全之间有紧密的联系。冗余性和多样性的使用, 对于达到可靠性是很重要的, 意味着系统能够抵抗那些瞄准专门设计和实现特性的攻击并从中恢复。支持高级别可用性的机制可以帮助系统从所谓的拒绝服务攻击中恢复过来, 这些攻击中攻击者的目标是使系统陷入瘫痪并让它停止正常工作。

设计一个系统要保证其安全, 不可避免地要有所取舍。为系统设计融入多种信息安全方案是完全有可能的, 这样会降低攻击成功的可能性。然而, 信息安全方案通常要求大量的额外计算并且影响系统的整体性能。例如, 我们可以通过加密机密信息来降低其被随便打开的可能性, 但是, 这就意味着信息的用户也必须等待信息解码, 这可能会减缓他们的工作。

信息安全和系统可用性之间也有制约关系。信息安全方案有时要求用户记住并提供附加信息 (例如, 大量的密码)。但是, 有时用户会忘记这些信息, 所以附加的信息安全意味着他们不能方便地使用系统。因此设计者必须在信息安全、系统性能和可用性之间找到一个平衡点。这将取决于系统的类型和它会被使用在什么地方。例如, 在一个军事系统中, 用户会对高级别信息安全系统很熟悉, 因此愿意接受并按着要求的频繁检查的流程进行。但是, 在一个股票交易系统中, 信息安全检查操作的中断将会是完全不可接受的。

14.2.1 体系结构设计

如第11章中所讨论的, 体系结构的选择对系统特性有深刻的影响。如果使用不恰当的体系结构, 维护系统信息的信息安全性和完整性或是保证系统有一定的实用性就是不可行的了。

在设计能维护信息安全性的系统体系结构时, 我们需要考虑两个基本问题:

1. 保护。如何组织系统使其关键资产能在外部攻击时得到保护?
2. 分布。如何对系统资产进行分布使得逞的攻击数减到最小?

这些问题总是存在冲突的。如果你将所有的资产放在一个地方, 那么就可以在其上建立多层保护。因为只要建立一个保护系统, 我们就能够为一个强大的系统提供多层保护。然而, 如果保护失败, 那所有的资产可就要受到危害了。增加若干个保护层也会影响系统的可用性, 所以这就意味着要达到系统可用性和性能的要求就更加困难。

另一方面, 如果你将资产分开存放, 保护的就会更高, 因为保护系统在每个备份中都要被实现。我们往往没有办法负担起这么多保护层的开支。保护伞被突破的几率也会大很多。但是, 如果真的发生攻击, 却不至于损失殆尽。将信息资产备份和分开是有可能的, 如果一个备份被毁坏或是不可存取, 那么其他的备份还可以使用。但是, 如果信息是保密的, 保留额外的副本

会增加入侵者获得这个信息的风险。

对于患者记录系统而言，使用集中式数据库系统是合适的。为了对系统提供保护，我们一般需要使用分层的体系结构，使关键性的受保护资产位于系统的最底层，并在其上设置各种不同的保护。图 14-4 是有关病人记录系统的说明，被保护的关键性资产是每个病人的记录。

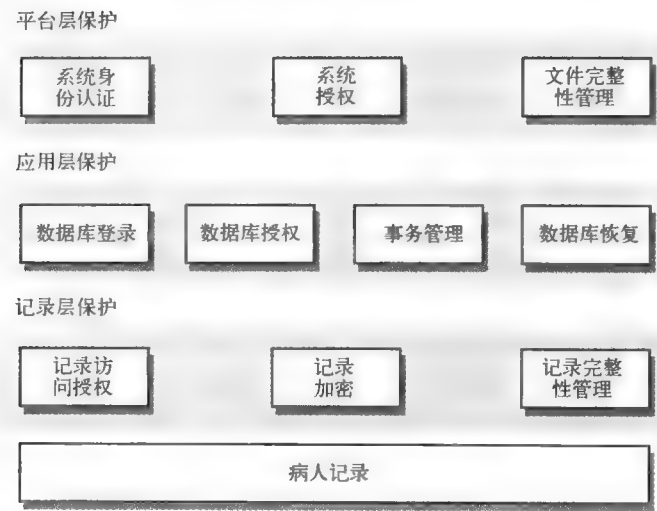


图 14-4 分层保护的体系结构

为了访问和修改病人的记录，攻击者需要穿越 3 个系统层：

- 1. 平台层保护** 最顶层控制对病人记录系统所在的平台的访问。这通常涉及用户到某个特别计算机的注册。平台也一般包括对系统上文件完整性维护的支持、备份等。
- 2. 应用层保护** 下一个保护层是建立在应用本身上的。它包括用户对应用的访问，用户身份认证以及对执行像浏览或修改数据等动作的许可。应用专门的完整性管理支持是可利用的。
- 3. 记录层保护** 当需要访问特殊记录且在此记录上执行所请求的操作时，对用户的授权检查调用这一级。此层的保护也可能包括加密来确保记录不会被文件浏览器所浏览。完整性检查，例如使用密码校验和，能检测出在正常记录更新机制之外产生的改变。

对任何一个应用来说，保护层的数量依赖于数据的危险程度。不是所有的应用都需要记录级的保护，因此，通常使用更大粒度的访问控制。为达到安全性，我们不能在每一层上使用相同的用户许可。理论上，如果是一个基于口令的系统，那么应用口令应该不同于系统口令也不同于记录级口令。但是，大量的口令对于用户记忆来说是困难的，并且他们会被重复的认证请求所激怒。因此，我们经常不得不考虑系统可用性而在信息安全上有所取舍。

如果对数据的保护是一个关键性需求，那么应该使用在服务器端包含有保护机制的客户 - 服务器体系结构。然而，如果保护失败，则攻击所带来的损失可能很高，恢复的成本也将很高（例如，所有用户的许可都需要重新颁发）。系统在拒绝服务攻击方面是脆弱的，拒绝服务攻击加重了服务器的负荷，使之不能让任何用户访问到系统数据库。

如果认为拒绝服务攻击是主要风险的话，我们可以决定让应用采用分布式对象体系结构。在此情形下，如图 14-5 所示，系统资产被分布到多个不同的平台上，每个平台有自己不同的保护机制。对某个节点的攻击可能造成某些资产是不可用的，但是还会有别的系统服务仍然保持可用。数据可以在系统的不同节点中复制，这样从攻击中恢复就简单了。

图 14-5 给出的是在纽约、伦敦、法兰克福以及香港市场上使用的银行股票和基金交易系统

的体系结构。它是一个分布式结构，每个市场的数据都是单独维护的。用于支持要求极高的股票交易（用户账目和价格）的关键活动所需的资产已经复制到各个节点上。如果系统某个节点受到攻击变得不可用时，紧要的股票交易活动能够转移到另一个国家或地区，这样对用户来说就是仍然可用的。



图 14-5 在证券交易系统中分布的资产

前面已经讨论过在信息安全和系统性能之间寻找平衡点的问题。很多时候，信息安全系统设计的一个问题是，能提供安全保障的最合适的体系结构可能不能达到最好的性能要求。例如，假设有一个应用其根本需求是维护某个大型数据库的机密性，对它的另一个需求是对数据能快速访问。高级别保护建议多个保护层是需要的，这就意味着系统的层次之间必须有通信。这就不不可避免地产生性能开销，从而降低数据访问的速度。如果采用另外的体系结构类型，那么实现保护和保证机密性会更加困难和需要更高的代价。在此情况下，你必须与系统客户讨论内在的冲突，在如何解决问题的方案上达成一致意见。

14.2.2 设计准则

并不存在一个如何达到系统安全性的硬性规定。不同类型的系统需要不同的技术措施来达到系统所有者所能接受的安全等级。不同用户组的态度和需求对什么是可以接受的以及什么是不能接受的有深深的影响。例如，对于银行来说，用户可能接受较高等级的安全性，因此比高校系统有更多的安全性程序。

然而，还是有一些一般性准则，广泛适用于系统信息安全性的方案设计，它们概括了信息安全系统的一些好的设计实践。信息安全性的一般性设计准则，例如所讨论过的，在这里有两个主

要的用途：

1. 它们提高软件工程团队对信息安全问题的重视。软件工程师通常只关注近期目标，如怎样让软件能用和能尽快交付给客户。信息安全问题很容易被忽视掉。这些准则能使信息安全问题在软件设计决策制定之时就得到考虑。

2. 它们能够在系统有效性验证过程中的复查清单中得到使用。从这里讨论的高层准则，我们可以导出更多探讨如何将信息安全设计到系统中去的详细问题。

10 个准则总结在图 14-6 中，这些准则有着不同的出处（Schneier, 2000；Viega 和 McGraw, 2002；Wheeler, 2003）。这里只讨论对软件描述和设计过程特别有用的准则，更一般性的原则，例如“保护系统中最脆弱的链接”、“保持简单”和“避免模糊”等都是重要的，但是它们并不直接与工程设计决策发生关系。

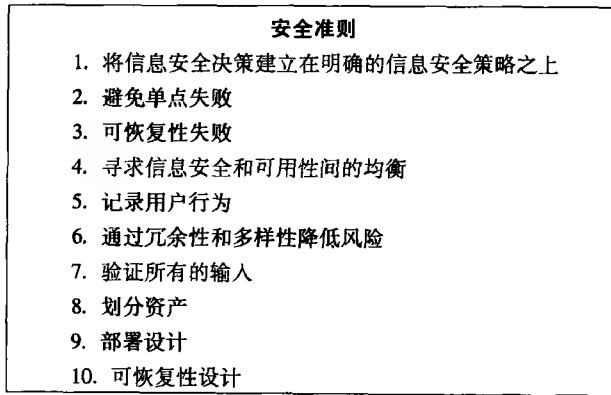


图 14-6 安全系统工程的设计准则

准则 1：将信息安全决策建立在明确的信息安全策略之上

信息安全策略是一个高层陈述，它给出了机构的基本信息安全条件。它要定义信息安全“是什么”而不是“如何达到”。此策略不应该规定提供和执行信息安全的机制。一般来讲，信息安全策略的各个方面都应该反映在系统需求当中。在实践中，尤其是使用快速应用开发过程的时候，做到这一点是不太现实的。因此，设计者就应该探讨信息安全问题，因为它给出了制定和评估设计决策的框架。

假如你在为本章前面提到过的 MHC-PMS（心理健康护理信息系统）设计访问控制系统。医院的信息安全策略是只有认可的临床工作人员可以修改病人的电子记录。因此你的系统必须包括检验每一个试图修改信息的人的合法性的机制以及拒绝无资格人员的非法修改的机制。

你可能遇到的问题是很多机构没有明确的系统信息安全策略。随着时间的推移，系统可能已经为了解决发现的问题做了更改，但是并没有详细的策略文档来指导系统进化。在这样的情况下，你需要从例子中制定出策略并写成文档，然后让公司管理者对它进行确认。

准则 2：避免单点失败

在任何要求极高的系统中，努力做到避免单点失败是个好的设计经验。这意味着系统某部分的单个失败不会造成系统的总体失败。用信息安全的术语，这意味着我们不应该依赖一个单一机制来确保信息安全，而是应该使用多个不同技术。有时我们称之为“深度防卫”。

例如使用口令来鉴别系统的用户，你也要包括口令/响应验证机制使得用户必须预先在系统中注册问题和答案。经过口令鉴定，用户必须正确回答问题方可进入系统。为保护系统中数据的完整性，你应该保存一份有关所有对数据修改的可执行日志（见准则 5）。以便在发生失败时，可以根据日志重新进行数据设置。你可能还要在变更发生之前对所有被修改了的数据进行备份。

准则 3：可恢复性失败

在所有系统中系统失败都是不可避免的，对于所有的系统都是这样，如同可恢复性失败之于安全要求极高系统，信息安全要求极高的系统也必须总是“失败可恢复的”。在系统失败时可使用的后退方式不能有比系统本身更差的安全性。系统失败也不能让攻击者访问到一般情况下不可能访问到的数据。

例如，在病人信息系统中，作者建议一个需求，即病人的数据应该在病人一进入诊所那一刻就下载到系统客户端。这样做能加速访问且能在服务器无法连接的时候仍可以访问到。通常，当病人看完病时服务器就删除此部分数据。然而，如果服务器失败，就存在信息被保留在客户端的可能。在这些情况下的可恢复性失败方法可以是对客户端病人数据进行加密。这就意味着未经授权的人不会读到这些数据。

准则 4：寻求信息安全和可用性间的均衡

信息安全要求和可用性要求有时会发生冲突。为了让系统安全，你就必须引入多个检查，包括确保用户获准使用系统的检查以及确保他们的行为遵守了信息安全策略。所有这些不可避免地要对用户施加要求——他们可能需要记住登录名字和口令，只能从某一台计算机上使用系统等。这些意味着用户在启动系统和有效地使用上需要更多的时间。当你添加这些信息安全特性到系统中的时候，不可避免地要降低系统的可用性。作者推荐过 Cranor 和 Garfinkel 的书 (2005)，其中对一般领域中的信息安全和可用性的广泛问题进行了讨论。

添加新的信息安全特性到系统中是要付出可用性代价的，这里存在一个平衡点，越过这个平衡点再增加信息安全特性就会达不到预期的目标。例如，如果你要求用户输入多个口令或者是需要频繁地更换口令，势必造成用户无法在脑中记住，他们可能就自然地会抄录口令在本子上。攻击者（尤其是内部人员）就很容易发现此口令并由此访问系统。

准则 5：记录用户行为

如果可以，我们应该始终保存一个用户行为记录。这个日志至少应该记录谁做了什么，使用的资产，以及行为的时间和日期。如准则 2 中所提到的，如果我们把这个保存为一个可执行命令表，那么我们就可以在失败后重新执行记录来恢复系统。当然，我们也需要工具来分析记录并且检测可能异常的行为。这些工具能够扫描记录并找出异常行为，因此可以帮助检测攻击和跟踪攻击者是如何获得系统使用权的。

除了帮助系统从失败中恢复以外，用户行为记录作为一个对内部攻击行为的威慑，也起到了很大作用。如果人们知道他们的行为会被记录，那么他们就不太可能做一些未经授权的事情。这对于偶然性攻击是很有效的，像一名护士查找患者记录，或是检查攻击通过社会工程盗取合法用户证书的地方。当然，这并不十分简单，作为技术上相当熟练的内部人员来说，也能够获得并修改记录。

准则 6：通过冗余性和多样性降低风险

冗余性就需要在系统中维护多于一个版本的软件和数据。多样性对于软件来说，意味着不同版本不应该基于相同的平台或者是使用相同的技术。因此，平台或技术漏洞将不会影响所有版本也就无法带来同样的失败。第 13 章中介绍了冗余性和多样性怎样作为基本的机制在可依赖性工程中使用。

我们已经讨论过冗余性的例子——第一个是在心理健康系统中维护服务器上的和客户端上的病人信息，第二个是分布式股票交易系统，如图 14-5 所示。在病人记录系统中，可以在客户端和服务端使用不同操作系统（例如 Linux 作为服务器端，Windows 作为客户端），这样确保基于操作系统漏洞的攻击无法同时影响服务器和客户端。当然，我们必须接受在一个机构中维护不同操作系统这样增长的管理开支作为获得这些好处的代价。

准则 7：验证所有输入

通常对系统的一个攻击总是通过一些未预料到的输入让系统以一种无法预知的工作方式工作。这些会直接造成系统崩溃，导致无法服务或者是输入会混合进恶意代码。缓冲区溢出漏洞，最早出现在互联网蠕虫（Spafford, 1989）中且常常被攻击者采用（Berghel, 2001），就是使用长输入串触发的。所谓的“SQL 中毒”，既恶意用户输入一个 SQL 片段让服务器解释，是另一个相当普遍的攻击。

正如第 13 章中介绍的，如果我们将输入有效性验证设计到系统中的话就能够避免很多这样的问题。基本原则是，永远不要未经任何检查就接受输入。作为需求的一部分，我们要定义应该应用的检查。应该使用输入的知识来定义这些检查。例如，如果要输入姓，就应该检查保证其中没有空格，且唯一可用的标点符号是连字号。同时还要检查所输入字符数并拒绝明显过长的输入。例如，没有谁的姓超过 40 个字符也没有一个住址会多于 100 个字符。正如第 16 章中讨论的，如果你用菜单呈现允许的输入，你就会避免输入有效性验证的很多问题。

准则 8：划分资产

分割意味着不用提供对系统中所有信息的访问权限也不会出现任何信息都无法访问的问题。而是将系统中信息合理地划分到一些仓位，从而使得用户只能访问到他/她所需要的那些而不是全部信息。这意味着攻击的影响被局部化。有些信息会丢失或者被损坏但是不太可能造成对系统中所有信息的全面影响。

举例，在病人信息系统中，所做的设计应该是这样的，在任何一个诊所中，诊所工作人员一般只能访问到预约此诊所的病人的记录。他们一般不应该看到系统中所有的病人的信息。这不仅会有利于限制由于可能的内部攻击造成损失，也意味着如果侵入者窃取了他们的证书，那么所造成的损失也是有限的。

说到此，我们还应该让系统有这样一种允许未预料访问的机制，即假设有病情十分严重的病人需要紧急处理而无需事先预约。在此情况下，我们应该使用另外的安全机制来克服系统中的信息分割。在这样的情况下，信息安全会被放宽来维持系统的可用性，使用一个登录机制来记录系统使用是必要的。然后我们能检查记录来跟踪任何未经授权的使用。

准则 9：部署设计

很多信息安全问题的发生是因为当系统在其运行环境中部署的时候没有能正确地配置。因此系统设计应该总是包含在用户环境中简化部署的一些设施，并检查潜在的配置错误和遗漏。这是一个很重要的话题，将在 14.2.3 节单独阐述。

准则 10：可恢复性设计

无论在系统安全维护上花多大的力气，都应该总是假设信息安全失败是要发生的。因此，我们应该考虑如何从可能的失败中恢复，并重建系统使之处于安全的运行状态。例如你会包含一个备份的验证系统以备你的口令认证被攻破。

假设某个未经批准的人从医院外部获得了对病人记录系统的访问权限，且我们不知道他们是如何获得有效登录名和口令的。我们需要重新初始验证系统，并且不仅仅是修改被侵入者使用过的证书。这个很有必要，因为侵入者可能也获得了其他用户的口令。因此，我们也必须改变所有认可用户的口令信息，以确保未经授权的用户没有办法进入口令变更机制中来。

因此，需要设计系统使其拒绝对没有更改口令的所有用户的访问直到他们已经完成了口令的修改为止，并认证真正用户以允许他们更改口令，尽管假设他们选择的口令可能是不安全的。一个方法是使用口令/响应机制，即用户必须回答他们事先注册的问题。这只有在修改口令发生改变的情况下被调用，使得系统从攻击中恢复过来，允许相对少量的用户受到影响。

14.2.3 部署设计

对系统的部署涉及诸多方面，包括配置软件使之适应运行环境，安装系统到此环境中的计算机上，然后是为这些计算机配置所安装的系统（见图14-7）。配置可以简单到仅仅是设置软件内置的某些参数来反映用户的偏好，也可能复杂到定义业务模型和规则，这些业务模型和规则支配软件的执行。

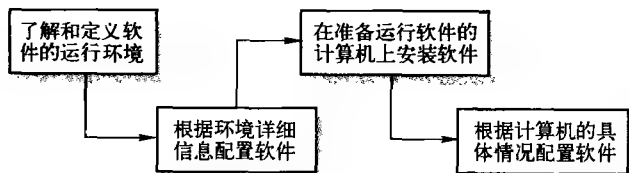


图 14-7 软件部署

就是在这个软件过程阶段中，通常会意外地导入软件漏洞。例如，在安装过程中，软件通常需要在配置中给出一个允许用户的列表，此列表仅仅包括普通管理员注册（例如“admin”）和缺省的口令（例如“password”）。这使得系统管理员设置系统非常容易。他们的第一个动作应该是给出一个新的注册名和口令并删除普通注册名。然而，很容易忘记这一点。知道缺省注册信息的攻击者就可以获得特殊权限访问系统。

人们通常将配置和部署看成是系统管理问题，而没有将其看成是软件工程过程范围内的事情。诚然，好的管理实践会避免由于配置和部署上出现的错误而发生很多安全问题。然而，软件设计者有责任去“为部署而设计”。我们应该总是提供内嵌的对部署的支持，此支持能降低系统管理者（或者用户）在配置软件过程中犯错误的几率。

作者推荐4种能在系统中加入部署支持的方法：

1. **包含对配置的观察和分析的支持** 应该总是在系统中包含让管理者或者是获准用户检查当前配置的设施。很奇怪的是此设施对于绝大多数系统来说是没有的，用户需要克服很多困难来寻找配置设定。例如，在作者写这一章时使用的文字处理器，不可能在一个屏幕上看到或打印出所有关于你的偏好的设置。但是，如果管理员能够得到配置的完整视图，他们就更容易发现错误和遗漏。理想情况下，配置显示也应该高亮显示出那些潜在在不安全的配置方面，例如，口令没有设置。

2. **最小化缺省特权** 所设计的软件应该让系统的缺省配置提供最小的、基本的特权。由此任何攻击者所能带来的灾难都能得到有效地控制。例如，缺省的系统管理员身份应该只允许访问让系统管理员重新设置证件的程序。它不应该允许访问任何其他系统设施。一旦新的证件已经建立，缺省的注册名和口令应该自动被删除。

3. **局部化配置设定** 在设计系统配置支持时，应该确保配置中的每一处如果是属于系统的相同一部分则设置在相同的位置。还是以微软 Word 为例，在作者所使用的 Word 版本中，可以设置某些安全信息，例如像使用“参数选择/安全”菜单设置文档访问控制的口令，而其他信息是在“工具/保护文档”菜单中设置的。如果设置信息没有得到局部化，很容易忘记设置了此项，或者是，在某些情况下，都不知道系统中包含某些安全设施。

4. **提供修补安全漏洞的简单方法** 应该包含简单的更新系统机制来修补已经发现的安全漏洞。这些可能包括自动检查安全更新和尽快下载这些更新。用户不能绕开这些机制，这是很重要的一点，因为不可避免地他们会认为其他工作会更重要。有多个有记录的严重的信息安全问题的例子表明，这些严重的信息安全问题的发生（例如，医院网络的彻底瘫痪）是由于用户在被要求更新软件时不去更新他们的软件。

14.3 系统生存能力

至此，已经从应用开发的角度讨论了信息安全工程问题。系统获得者和开发者对有可能受到攻击的系统的所有方面都能够进行控制。现实情况是，如图 14-1 中所暗示的，现在的分布式系统不可避免地依赖于基础设施，包括商业现货系统和可复用组件，这些商业现货或者是组件都是分别开发的。这些系统的信息安全特性不仅仅依靠于局部设计决策，它同样受到外部应用的信息安全、Web 服务和网络基础设施的影响。

这意味着，无论我们对信息安全投入多少关注，都无法保证系统能抵御外部攻击。因此，对于复杂的网络系统，应该假定渗透是无处不在的，系统完整性是无法得到保证的。由此，我们应该考虑如何将系统做得更有弹性，使之在最恶劣的情况下也照样能提供给用户最基本的服务。

生存能力（Westmark, 2004）是系统的作为一个整体所表现出来的总体特性，而不是单个组件的特性，不是各个组件的生存能力。系统生存能力反映了它在受到攻击甚至是在被攻击后，或者系统失败发生、系统部分遭到毁坏后，仍然能够连续提供给合法用户基本业务或关键任务服务的能力。

我们致力于系统生存能力是因为我们的经济和社会生活都依赖于计算机控制的要求极高的基础设施。这包括供给（如电力、水、气等）基础设施，同样重要的还有交通和电信（如电话、因特网、邮政服务等）基础设施。然而，生存能力不仅仅是要求极高的基础设施问题，任何依赖于要求极高的网络计算机系统的机构都应该关心他们的业务在系统无法抵挡恶意攻击或者是发生灾难性系统失败的情形下会受到多大的影响。因此，对于业务要求极高的系统，生存能力分析和设计应该是信息安全工程过程的一个组成部分。

维护要求极高的服务的可用性是生存能力的基本内容。这意味着我们需要了解：

- 业务当中最为关键的系统服务；
- 需要维持的最低的服务质量；
- 这些服务怎样得到折中；
- 如何能保护这些服务；
- 如果服务不再可用的话，如何迅速恢复。

例如，在救护车调度系统中，要求极高的服务是那些与接听电话和为紧急救治派遣救护车相关的服务。其他服务例如电话记录，救护车位置管理也是次等重要的，因为它们不要求实时处理或者是因为其他机制可以代替使用，例如，为找到救护车的位置，我们可以呼叫救护车上的人询问他们所在的位置。

Ellison 和 Colleagues (1999a, 1999b, 2002) 设计了一个分析方法，被称为可生存系统分析。它是用来评估系统中的漏洞并支持对系统体系结构和能提升系统生存能力的特征的设计。他们的论点是获得生存能力依赖于 3 个互补的策略：

1. **抵抗力** 通过构建击退攻击的能力避免出现问题。例如，系统可以使用数字证书来鉴别用户，这样使未经认定的用户获得访问能力更加困难。
2. **识别** 通过构建检测攻击和失败并评估损失后果的能力来检测问题。例如，检验码可以加入到关键数据中，这样对数据的损坏就能够得到检测。
3. **恢复** 通过构建在受到攻击时提供基本服务的能力和在攻击发生后恢复完全功能性的能力渡过困难。例如，可以包括对同一功能使用不同实现的容错机制，以应对部分系统服务的缺失。

可生存系统分析是个 4 阶段过程（见图 14-8），分析当前的或被提议的系统需求和体系结构，识别关键性服务、攻击想定和系统“弱点”，以及提出变更来改善系统的生存能力。在每个阶段的关键活动是：

- 1. **系统理解** 对一个已经存在的或者是准备建立的系统，复查系统的目标（有时也称为游戏目标）、系统需求和系统体系结构。
- 2. **关键服务识别** 找出必须维护的服务和用于维护这些服务所需要的组件。
- 3. **攻击仿真** 找出对于可能的攻击的想定或用例和这些攻击会影响的系统组件。
- 4. **生存能力分析** 找出那些基本的组件和易受攻击的组件，找出基于抵抗性、识别和恢复的生存能力策略。

Ellison 和他的同事提出了此方法的一个极好的用例研究，是基于支持心理健康治疗系统的（1999b）。这个系统与本书中的 MHC-PMS（心理健康护理信息系统）例子很类似。这里不想简单重复他们的分析，而是使用股票交易系统（如图 14-5 所示）来解释生存能力分析的某些特征。

从图 14-5 中可以看到，此系统已经有了某些生存能力方面的考虑。用户账号和股票价格在服务器间复制，这样当服务器无法使用时股票认购仍能够进行。让我们假设认可用户的下单能力是必须维护的关键服务。为了确保用户对系统的信任，对系统完整性的维护是基本的。订单必须是精确的，且反映的是系统用户所作出的实际交易。

为维护此订购服务，使用了 3 个系统组件：

- **用户身份验证** 它允许经认可的用户对系统的登录。
- **行情表** 它可以给出股票的买和卖的价格。
- **下单** 它允许在某个价位上的买单和卖单能立即得到执行。

这些组件显然使用了一些基本的数据资产，如用户账号数据库、价格数据库和订单交易数据库。如果要维持服务，它们必须能够在攻击中存活下来。

对此系统的攻击可以有多种类型。这里我们只考虑两种可能性：

- 1. 恶意用户嫉恨有资格的系统用户。他利用正当用户的证书获取对系统的访问。向系统发出恶意的订单，实现了股票的买卖，有正当权限的用户就会出现问题。
- 2. 未经授权的用户通过获取直接发出 SQL 命令的许可使得交易数据库毁坏。买和卖的关系因此无法实现。

图 14-9 给出了可能用于帮助反击这些攻击的抵抗、识别和恢复策略。

攻 击	抵 抗	识 别	恢 复
未经授权用户的恶意下单	要求一个交易口令用于下单，此口令要不同于登录口令	将订单复印件通过电子邮件发送给授权用户，同时包括联系电话（这样可以检测恶意下单） 维护用户的订单历史并检查非正常的交易模式	提供自动“撤销”交易的机制并存储用户的账户 在有恶意交易发生时退还用户的资金。保证不再会发生其他损失
交易数据库崩溃	要求特权用户通过更强的认证机制（例如数字证书）来认证	在国际服务器上为办公室维护只读交易拷贝。定时地比较交易检查其中的问题 对所有交易记录维护密码的校验码以检查其中的问题	从备份拷贝中恢复数据库 提供一个在指定的时间上重放交易的机制以创建交易数据库

图 14-9 股票和基金交易系统生存能力分析

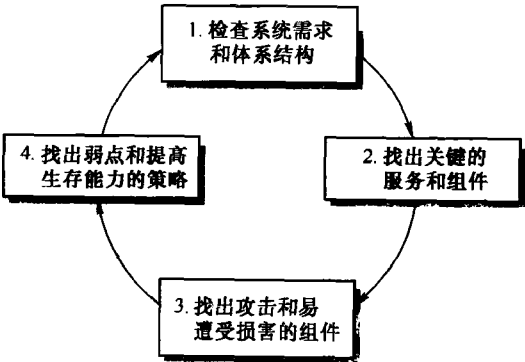


图 14-8 生存能力分析的各个阶段

当然，提高系统的生存能力和恢复能力是需要付出代价的。公司如果从来没有受到过严重攻击或者没有产生过相关损失，可能就不情愿在提高生存能力方面投资。然而，就像我们平时未雨绸缪而不是亡羊补牢，最好是在没有发生成功攻击之前对生存能力做投资。生存能力分析还不是绝大多数软件工程过程的一个部分，但是，随着越来越多的系统成为业务要求极高的系统，这样的分析似乎得到了更为广泛的使用。

要点

- 信息系统安全主要关注如何开发和维护软件系统，使之能抵御那些意在破坏基于计算机系统或其数据的恶意攻击。
- 信息安全威胁是对系统或其数据的机密性、完整性或可用性的威胁。
- 信息安全风险管理包括对攻击产生的损失的评估，导出意在消除或减少这些损失的信息安全需求。
- 信息安全的设计包括：对安全系统体系结构的设计，遵从好的安全系统设计经验，以及加入某些能尽可能减少在系统部署中导入安全脆弱性可能的功能。
- 在设计安全系统体系结构时的关键问题包括：组织系统结构来保护关键性资产，分布存储系统资产以最小化成功攻击所带来的损失。
- 信息安全设计准则能使系统设计者对先前可能没有考虑到的安全问题更加敏感。它是创建安全复查清单的基础。
- 为支持安全部署，应该提供显示和分析系统配置的手段，使配置设置局部化以便不会忘记重要的配置，最小化系统用户的缺省特权，并提供修复信息安全脆弱性的手段。
- 系统生存能力反映的是系统在受到攻击或者是部分系统受到损坏后持续向合法用户提供基本业务或任务要求极高的服务的能力。

进一步阅读材料

《Survivable Network System Analysis: A Case Study》这是一篇非常棒的介绍系统生存能力的论文，使用对心理健康记录处理系统案例分析来阐述生存能力方法的应用（R. J. Ellison, R. C. Linger, T. Longstaff and N. R. Mead, IEEE Software, 16 (4), July/August 1999）。

《Building Secure Software: How to Avoid Security Problems the Right Way》这是一本很好的实用类书，从编程视角介绍所看到的所有信息安全方面问题（J. Viega and G. McGraw, Addison-Wesley, 2002）。

《Security Engineering: A Guide to Building Dependable Distributed Systems》这是一关于构建安全系统问题的全面而又深入的讨论。它是着眼于系统而非单独的软件工程，全面涉及了硬件和网络。还有非常棒的来自真实系统失败的例子（R. Anderson, John Wiley & Sons, 2001）。

练习

- 14.1 解释应用信息安全工程和基础设施信息安全工程之间的重要区别。
- 14.2 对于 MHC-PMS（心理健康护理信息系统），给出有关资产、失去保护、脆弱性、攻击、威胁和控制的例子。
- 14.3 解释为什么有必要让风险评估成为一个从早期的需求工程阶段到运行使用阶段都持续的过程。
- 14.4 利用对 14.2 题有关 MHC-PMS 系统的回答，评估与此系统相关联的风险，并提出两个能降低这些风险的系统需求。
- 14.5 利用一个非软件工程的上下文的类比，解释为什么应该利用分层方法于资产保护。

- 14.6 解释为什么在系统可用性是非常关键的场合使用多种技术来支持分布式系统是重要的。
- 14.7 什么是社会工程？为什么在大机构中避免它是困难的。
- 14.8 对任何我们在使用的商业现货软件系统来说（例如微软的 Word），请分析其所包含的配置工具并讨论其中你发现的问题。
- 14.9 解释如何综合使用抵抗、识别和恢复策略来增强系统的生存能力。
- 14.10 对于证券交易系统，如在 14.2.1 节中讨论的和在图 14-5 中给出的，提出两个更为可信的对系统的攻击，并提出对抗这些攻击的可能策略。

参考书目

- Alberts, C. and Dorofee, A. (2002). *Managing Information Security Risks: The OCTAVE Approach*. Boston: Addison-Wesley.
- Alexander, I. (2003). 'Misuse Cases: Use Cases with Hostile Intent'. *IEEE Software*, 20 (1), 58–66.
- Anderson, R. (2008). *Security Engineering, 2nd edition*. Chichester: John Wiley & Sons.
- Berghel, H. (2001). 'The Code Red Worm'. *Comm. ACM*, 44 (12), 15–19.
- Bishop, M. (2005). *Introduction to Computer Security*. Boston: Addison-Wesley.
- Cranor, L. and Garfinkel, S. (2005). *Security and Usability: Designing secure systems that people can use*. Sebastopol, Calif.: O'Reilly Media Inc.
- Ellison, R., Linger, R., Lipson, H., Mead, N. and Moore, A. (2002). 'Foundations of Survivable Systems Engineering'. *Crosstalk: The Journal of Defense Software Engineering*, 12, 10–15.
- Ellison, R. J., Fisher, D. A., Linger, R. C., Lipson, H. F., Longstaff, T. A. and Mead, N. R. (1999a). 'Survivability: Protecting Your Critical Systems'. *IEEE Internet Computing*, 3 (6), 55–63.
- Ellison, R. J., Linger, R. C., Longstaff, T. and Mead, N. R. (1999b). 'Survivable Network System Analysis: A Case Study'. *IEEE Software*, 16 (4), 70–7.
- Pfleeger, C. P. and Pfleeger, S. L. (2007). *Security in Computing, 4th edition*. Boston: Addison-Wesley.
- Schneier, B. (2000). *Secrets and Lies: Digital Security in a Networked World*. New York: John Wiley & Sons.
- Sindre, G. and Opdahl, A. L. (2005). 'Eliciting Security Requirements through Misuse Cases'. *Requirements Engineering*, 10 (1), 34–44.
- Spafford, E. (1989). 'The Internet Worm: Crisis and Aftermath'. *Comm ACM*, 32 (6), 678–87.
- Viega, J. and McGraw, G. (2002). *Building Secure Software*. Boston: Addison-Wesley.
- Westmark, V. R. (2004). 'A Definition for Information System Survivability'. 37th Hawaii Int. Conf. on System Sciences, Hawaii: 903–1003.
- Wheeler, D. A. (2003). *Secure Programming for Linux and UNIX HOWTO*. Web published: <http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html>.

可依赖性与信息安全保证

目标

本章的目标是讨论在要求极高的系统的开发中使用的检验和有效性验证技术。读完本章，你将了解以下内容：

- 理解不同的静态分析方法在要求极高的系统的检验中怎样使用；
- 理解可依赖性和信息安全性测试中的基本要素，以及测试要求极高系统的固有问题；
- 了解过程保证为什么是重要的，特别是当软件需要由一个管理部门认证时；
- 介绍了安全和可依赖性用例，给出系统安全和可依赖性的论点和证据。

可依赖性和信息安全性保证是关于对要求极高的系统能否满足其可依赖性要求的检查。这要求检验和有效验证（V&V）过程查找需求说明、设计以及程序错误等可能影响可用性、信息安全性、可依赖性和系统安全的因素。

对要求极高的系统的检验和有效性验证显然与其他系统的有效性验证是很相似的。其 V&V 过程应该表明系统满足它的描述，而且系统所提供的服务和所表现出来的行为能满足客户的需求。这样做，通常能够表明需求，并且发现需要修补的设计错误和程序漏洞。不过，要求极高的系统要求特别严格的测试和分析，原因有两个：

1. **失败的代价** 要求极高的系统失败的潜在代价和后果比一般系统要高得多。要花更多的时间降低系统失败的可能性，通常情况下系统发布前找出和纠正错误付出的成本要低于系统维护中系统出现意外所付出的代价。

2. **可依赖性属性的有效性验证** 客户和管理部门需要确信所定义的可依赖性属性（可用性、可依赖性、安全性和信息信息安全性）已经得到满足。在某些情况下，外部管理部门如民用航空权威机构必须在其部署前确保系统的安全。要获得认证，需要展示系统怎样通过有效性验证。要做到这一点，就必须设计和执行特殊的 V&V 程序来收集有关系统可依赖性的证据。

因为这些原因，要求极高的系统的 V&V 成本通常比其他类型系统要高得多，往往占总开发成本的一半以上。

尽管 V&V 的成本很高，但是由于它们常常能使故障发生的损失明显降低，所以也被认为是值得的。举例来说，在 1996 年，阿丽亚娜 5 号火箭上的软件系统失败，损失了多个卫星，尽管没有导致人员伤亡，但是造成的间接经济损失达到数亿美元。随后的调查发现系统检验和有效性验证的不足部分导致了失败。如果使用了更有效的审查，往往也是相对代价低的，可能已经发现了导致事故的问题。

虽然要求极高的系统的有效性验证过程应该主要集中在确认系统的有效性和信息安全性保证上，但也应该有相关的活动来确认系统开发所采用的过程是可靠的。如第 13 章中说明的，系统开发过程的质量严重影响系统的质量。简言之，一个好的过程能产生一个好的系统。

可依赖性和信息安全性保证过程的结果是一个有形的看得见的证据，比如说关于系统可依赖性的审查报告、测试结果等。这样的证据可能后来会用来证明关于部署和使用一个系统是足够可靠的和安全的这样一个决定。有时，系统可依赖性的证据体现在一个可依赖性和安全性的用例中。用这样的用例来说服客户和外部管理部门，开发者在系统可依赖性 or 安全性方面的信

心是有理由的。

15.1 静态分析

静态分析技术是不涉及程序执行的系统检验技术。它工作在软件的原始形态层面上,比如描述模型和设计模型,或者是程序的源代码。静态分析技术能够用来检查系统的描述和设计模型,从而在系统的可执行版本之前发现错误。它的优势在于其错误呈现不会干扰系统的检测。当测试一个程序时,一个缺陷可能掩盖其他缺陷,所以当删除一个检测到的缺陷时必须重复测试程序。

正如第8章中所讨论的,可能最通常使用的静态分析技术是同行评审和检查:描述、设计和程序是由同一组人检查的。他们详细地检查设计和代码,查找可能的错误和遗漏。另一个技术是使用设计建模工具来检查统一建模语言(UML)中的异常,比如说在不同的对象中名称的重复使用。然而,对于要求极高的系统来说,下列静态分析技术可能会被用到:

1. 形式化的检验,即用严密的数学论证以说明程序符合它的描述说明。
2. 模型检测,即用一个定理证明器来检查系统的形式化描述的 inconsistency。
3. 自动程序分析,即检查程序源代码的形式,以发现其可能存在的错误。

这些技术之间是紧密联系的。模型检测依赖于系统的一个形式化模型,该模型创建自形式化描述。静态分析器可能使用嵌入在程序中的形式化断言作为注解来检查相关代码与这些断言的一致性。

15.1.1 检验和形式化方法

正如第12章中所讨论的那样,软件开发的形式化方法是基于作为系统描述的一个系统形式化模型的。这些形式化方法主要关注于对描述的数学分析;将这个描述转变成成一个更详细的、语义上等价的描述;或者使用形式化检验来验证系统的一种描述和另一种描述在语义上是等价的。



净室开发

净室软件开发是建立在形式化软件检验和静态测试的基础上的。净室处理的目标是零缺陷软件,用来保证发布的系统有高级别的可依赖性。在净室过程中,每一个软件增量被形式化地详细定义,这种详细的描述接着被转换为实现。软件的正确性通过使用一种形式化的方法来证明。在这个过程中没有对缺陷的单元测试,并且系统测试是以评估系统的可依赖性为中心的。

<http://www.SoftwareEngineering-9.com/Web/Cleanroom/>

形式化方法可以在 V&V 过程的不同阶段中使用:

1. 系统的形式化描述的使用和数学分析可能是为了一致性。这种技术对发现描述中的错误和遗漏是有效的。模型检测是一种描述分析,下一章节会讨论到。
2. 使用数学证明,可以形式化检验软件系统的代码和它的描述是一致的。这就要求有一种形式化的描述且是对发现程序错误和某些设计错误是有效的。

因为形式化的系统描述和程序代码之间存在较大的语义上的差距,所以很难证明一个单独开发的程序是和它的描述相一致的。因此,程序检验的工作现在是基于变换开发的。在一个变换开发过程中,一个形式化的描述通过一系列的程序代码表示。软件工具支持变换开发,并且帮助

核实相应系统表示的一致性。B 方法很可能是最为广泛使用的形式化的变换方法（Abrial, 2005; Wordsworth, 1996）。这种方法已经被用于开发火车控制系统和航空电子技术软件。

形式化方法的支持者声称使用这些方法会产生更加可靠和安全的系统。形式化检验论证开发的程序符合它的描述并且其实现中的错误不会危害系统的稳定性。如果像用 CSP（Schneider, 1999）这样的语言写的描述来开发并发系统的一个形式化的模型，可能会发现那些导致最终程序死锁的情况，并且能够解决这些问题。这是很难仅仅通过测试来完成的。

然而，形式化描述和证明并不能保证软件在实际使用中将是可靠的。其原因包括：

1. 描述可能没有反映系统用户真正的要求。这如第 12 章中所提到的，系统用户几乎不可能理解形式化符号系统，所以他们不能通过直接阅读形式化文档来找到错误和遗漏。这就意味着形式化的描述极有可能包含错误，并且不是系统需求的一个准确表示。

2. 证明可能包含错误。程序证明是庞大和复杂的，所以，像庞大和复杂的程序本身一样，它们通常也包含错误。

3. 证明所假设的使用方式可能是错误的。如果系统不按假设的情况使用，那么证明就是无效的。

检验一个重要软件系统需要很多时间，还需要数学专家和专门的工具，如定理证明器。因此这是一个极端昂贵的过程，随着系统规模的增加，形式化检验的花费不成比例地增加。因此很多人认为形式化检验是不划算的。软件工程师们认为对系统的相同水平信任程度可以使用其他的有效性验证技术，例如审查和系统测试，而这要便宜得多。

尽管有这些不足，但是形式化方法在开发要求极高的软件系统过程中扮演着一个重要的角色（在第 10 章讨论过）。形式化描述在发现描述中的问题是十分有效的，这些问题通常导致系统失败。尽管形式化的检验对于大型系统来说仍然是不现实的，但是它能用来检验那些安全要求极高和信息安全要求极高的组件。

15.1.2 模型检测

使用演绎的方法形式化地检验系统是比较困难且昂贵的，人们又提出了另外一种形式化分析方法，它是基于一个更有限的正确性概念。这些方法中最为成功的是模型检测（Baier 和 Katoen, 2008）。这种方法被广泛用来检查硬件系统的设计，并且在像 NASA 的火星探测车（Regan 和 Hamilton, 2004）和电话呼叫处理软件这样的要求极高的软件系统中越来越多地使用。

模型检测包括创建一个系统模型，并且使用特殊的软件工具检查模型的正确性。已经开发出来了很多不同的模型检测工具——对软件来说，最为广泛使用的是 SPIN（Holzmann, 2003）。模型检测中的各个阶段如图 15-1 所示。

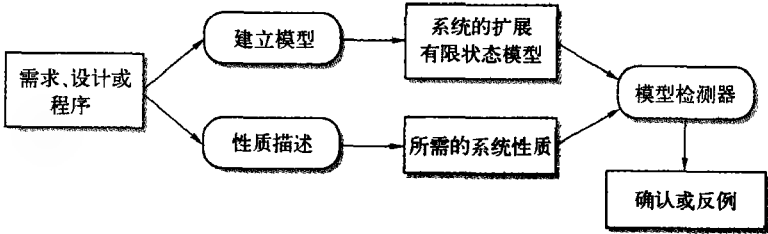


图 15-1 模型检测

模型检测过程包括建立系统形式化模型，这里通常要用到扩展有限状态机。模型检测系统使用什么语言，模型就被表达为什么的语言。例如，SPIN 模型的检查器使用 Promela 语言。找出一组需要的性质并写成形式化表达，它们都是基于时序逻辑的。例如，对于野外气象站系统中的

性质可能是,系统总是从“记录”状态到达“传送”状态。

模型检测器接着要在模型中探测所有路径(例如,所有可能的状态转换),检查每条路径所保持的性质。如果成功,模型检测器会确认这个模型关于这个属性是正确的。如果对于某条特殊的路径此性质不能保持,模型检测器就输出一个反例来说明性质在哪里出错了。模型检测在并发系统的有效性验证中特别有用,众所周知,这种验证由于系统对时间的敏感性很高而非常难以实施。检查器可以通过交叉探测和并发转换来发现潜在问题。

模型检测中的一个关键问题是系统模型的创建。如果模型需要人工建立(从一个需求或设计文档出发),这将是一个花费很高的过程并且会占用很多时间。另外,也有可能建立的模型不是需求和设计要求的精确模型。因此,如果模型能够从源代码开始自动地被建立是最好的选择。Java Pathfinder (Java 路径查找)系统(Visser 等,2003)是直接从 Java 代码开始模型检测的一个例子。

模型检测时计算开销很大,因为它使用一个详尽彻底的方法来检查所有通过系统模型的路径。随着系统大小的增加,状态也随之增加,结果是需要检查的路径数目也增加了。这就意味着,对于大型系统,模型检测可能是不实际的,原因是执行这些检查要耗费大量的计算时间。

然而,随着识别部分未探索过的状态的算法的改进,在要求极高的系统开发中例行使用模型检测变得更加可行。虽然它并不能真正适用于机构的面向数据的系统,但是它能够被用来检验嵌入式软件系统,此类系统建模为状态机模型。

15.1.3 自动静态分析

正如第8章中所讲的,程序审查通常是错误列表和启发法驱动的。它们可以识别各种编程语言中的一般性错误。对于某些错误和启发式,可以根据这些表自动生成对程序检查的过程。由此也导致了能够发现代码中错误片段的自动化静态分析器的开发。

静态分析工具对系统源代码进行加工,并且至少对于一些类型的分析不需要进一步的输入。这就意味着程序员不需要学习专门的符号来写系统描述,因此分析的好处就很显而易见了。这让自动静态分析比形式化的有效性验证和模型检测更加容易地引入开发过程。因此,它可能是静态分析技术中最为广泛使用的技术。

静态程序分析器是一个软件工具,扫描程序源代码,审查可能的缺陷和异常之处。它不需要去执行程序,而是通过解析程序文本从而识别出程序语句的各个部分。由此,分析器审查出语句是否合乎规则,推断出程序的控制结构,并计算出可能的程序变量值。静态程序分析器弥补了语言编译器提供的错误检测设施的不足。它们可以作为审查过程的一部分或者是作为单独的一个检验及有效性验证过程活动。自动程序分析比详细的代码审查更加快速和低成本。然而,它不能够发现在程序审查会中所指出的那几类错误。

自动静态分析的意图是引起代码阅读器对程序中的异常的注意,例如变量没有初始化、变量未曾使用、变量值越界等,由静态分析能审查出来的内容在图15-2中列出。当然,具体的语言需要具体的检查并且取决于该语言的约定。异常通常是程序错误和遗漏的结果,所以它们突显那些在程序运行时出错的一些东西。然而,我们应该明白,这些异常不一定是程序缺陷。它们可能是故意制作的,或者是不会产生不良后果的。

在静态分析器中,有3个级别的检验可能需要实现:

1. **特有错误检查** 在这个级别上,静态分析器知道程序员使用像 Java 或 C 这样语言编写的通常的错误。分析器分析代码,形式查找出特有的问题并且标记出来呈现给程序员。尽管相对简单,分析是基于普通错误的,但是却是非常成本有效的。Zheng 和他的合作者(2006)研究了在 C 和 C++ 语言的大型代码中使用静态分析,并且发现 90% 的错误是由 10 种特有错误造成的。

缺陷类	静态分析检查
数据缺陷	变量在初始化前被使用 变量声明了但未使用 变量赋值了两次，但两次赋值之间并未得到使用 可能的数组边界越界 未声明的变量
控制缺陷	不可到达的代码 无条件的转移到循环中
输入/输出缺陷	变量输出两次，其间并未有新的赋值
接口缺陷	参数类型不匹配 参数数量不匹配 函数结果无法使用 未被调用的函数和程序
存储管理缺陷	未赋值的指针 指针计算 内存泄漏

图 15-2 自动静态分析检查

2. 用户定义错误检查 在这种方法中，静态分析器的用户可能定义了错的格式，因此扩展类型的错误可以被检测到。在必须保持顺序的情况下（例如，A 方法必须始终在 B 方法之前被调用），这种方法显得特别有用。久而久之，机构可以收集发生在他们程序中的关于普通故障的信息，并且扩展静态分析工具来标出这些错误。

3. 断言检查 这是静态分析中最为常规和最强大的方法。开发者在程序中包含一个形式化断言（通常写成固定格式的注释），以声明在程序某个位置必须满足的关系。例如，断言可能声明一些变量的值必须在 $x \sim y$ 的范围内。这些分析器符号化地执行代码并突显断言不能保持的语句。这种方法使用在像 Splint（Evans 和 Larochelle，2002）和 SPARK Examiner（Croxford 和 Sutton，2006）这样的分析器中。

静态分析对查找程序中错误是很有效的，但是一般会产生大量的“误报”。本来没有错误的代码部分，但是按静态分析器的规则却检查出有错误的可能性。误报的数目可以通过用断言的形式对程序增加更多的信息来减少。当然，这需要代码开发者额外的工作来完成。在代码自己能够被检查出错误之前，对产生这些误报的筛选工作就要完成。

静态分析对于信息安全性检查来说尤为具有价值（Evans 和 Larochelle，2002）。静态分析器可以被用来检查众所周知的问题，比如说缓冲区外溢，还有未检查的输入，这些输入可能来自攻击者。对众所周知问题的检查对于提升信息安全性来说很有效，因为大部分攻击者将他们的攻击建立在这些普通的漏洞上。

信息安全性测试是比较困难的，因为攻击者常常会做出一些意料之外的事情，测试者是很难预测到的，这一点后面会讲的。静态分析器能将测试者可能不具备的复杂的安全专业知识加入其中，并且在程序被测试之前就可以应用。如果使用静态分析，你可以断言所有可能的程序执行都是正确的，而不仅仅是那些和你设计过的执行相符合的测试。

如今静态分析在很多机构中的软件开发过程中例行使用。微软在容易因为程序故障导致一系列严重影响的设备驱动器方面引进了静态分析（Larus 等，2003）。他们现在将这个方法扩展到更广泛的软件范围来查找信息安全性问题和影响程序可依赖性的错误（Ball 等，2006）。很多要求极高的系统，包括航空电子技术和核子系统，都例行地把静态分析作为 V&V 过程的一部分

(Nguyen 和 Ourghanlian, 2003)。

15.2 可靠性测试

可靠性测试是度量系统可靠性的一个测试过程。正如第 10 章中所解释的, 已经有一些可靠性量度指标, 像请求失败概率 (probability of failure on demand, POFOD) 和失败发生率 (the rate of occurrence of failure, ROCOF)。这些指标从量上详细说明了所需软件的可靠性。如果系统已经达到了要求的可依赖性水平, 可以在可靠性测试过程中得到验证。

测量系统可靠性的过程如图 15-3 所示, 这个过程包括 4 个阶段:

1. 从研究已经存在的同一类型系统开始, 理解这些系统在实践中是怎样被使用的。这很重要, 因为你必须通过系统用户的使用来度量其可靠性。目的是建立运行概况。运行概况要找出系统输入的分类以及这些输入将在正常使用中出现的可能性。
2. 构造一个能反映运行概况的测试数据集合。这就是说要获得具有相同概率分布的测试数据作为所研究系统的测试数据。通常可以使用测试数据生成器得到测试数据。
3. 使用上面生成的测试数据对系统进行测试、记录发现的失败和每个失败类型发生的次数。正如第 10 章所讨论的, 所采用的时间单位应该适合相应的可靠性度量。
4. 当观察到相当数量的失败后, 软件的可靠性就可以计算了。我们就可以计算出适当的可靠性度量值。

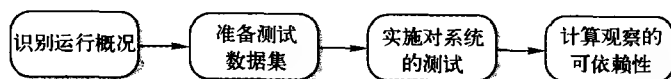


图 15-3 可依赖性度量

这个四步方法有时被称为统计测试。统计测试的目标是要评估系统的可靠性。这与缺陷测试 (第 8 章中讨论过的) 形成鲜明对比。缺陷测试是要发现系统的缺陷。Prowell 等 (Prowell 等, 1999) 在他们的有关净室软件工程的书中讨论了统计测试的各个方面。

这些可靠性测量方法虽然在理论上看起来很好, 但是实际应用过程中并不是那么简单。

主要的困难在于:

1. **运行概况的不确定性** 运行概况可能无法精确反映系统真实的使用情况。
2. **测试数据生成的高成本** 生成运行概况所需要的大量数据是非常昂贵的, 除非此过程完全能自动完成。
3. **在指定高可靠性情况下统计的不确定性** 需要生成相当数量的有统计意义的失败才能得到精确的可靠性测量结果。当软件已经可靠时, 发生的错误相对较少, 并且不容易产生新的错误。
4. **识别故障** 系统故障的发生并不总是很明显。如果你有一个形式化的描述, 也许能够从描述中确定偏差。但是, 如果描述是自然语言, 由于其可能具有二义性, 观察者可能在系统是否发生故障方面存在分歧。

到目前为止, 产生大测试数据集的最好方法是使用某些测试数据生成器, 这些测试数据生成器能自动生成运行概况中规定的输入类型的测试数据。然而, 对于交互式系统却不总是能自动产生出所有的测试数据, 因为输入通常是对系统输出的响应。因而数据需要由手工设计, 这项工作是费用很高的。但完全自动化也是不可能的, 而且编写测试数据产生器的命令又要耗费大量时间。

统计性测试可结合故障注入来收集数据以说明缺陷测试过程是怎么有效的。故障注入

(Voas, 1997) 是有意地在程序中植入错误。当程序执行时, 这些错误将导致程序出错并产生相关的故障。然后我们分析故障, 来发现问题的根源是否是你增加进程中的一个错误。如果发现 X% 的注入错误导致了故障, 那么故障注入的支持者会声称, 这表明缺陷测试过程也会在程序中发现 X% 的实际错误。

当然, 这是假设注入错误的分布及类型和实际中出现的错误是一致的。对由于编程错误所产生的缺陷来说, 这种假设应该是合理的, 但是故障注入在预测源于需求和设计缺陷而产生的错误数量来说并不有效。

统计性测试通常能发现那些 V&V 过程发现不了的错误。这些错误可能意味着系统的可依赖性不能达到要求并且必须进行修订。通过这些修订, 系统能够重新测试并再次评估其可依赖性。当修订和再测试过程重复进行若干次后, 推断结果并预测什么时候能达到要求的可依赖性级别就是可能的事了。我们首先要有一个可依赖性增长模型, 它描述了可依赖性是怎么随着时间改善的。然后找到推断数据和增长模型的匹配点, 这样有助于对测试做规划。有时, 一个增长模型可能揭示了所要求的可依赖性级别是不可能达到的, 所以需要重新协商需求。



可依赖性增长模型

可依赖性增长模型是关于系统的可依赖性随着测试工作的进行如何改变的。随着系统故障的暴露, 潜在的导致这些故障的错误被修订, 所以系统可依赖性在系统测试和调试期间得以提升。为了预测可依赖性, 概念上的可依赖性增长模型必须转换成一个数学模型。

[http://www. SoftwareEngineering-9. com/ Web/ DepSecAssur/ RGM. html](http://www.SoftwareEngineering-9.com/Web/DepSecAssur/RGM.html)

运行概况

软件的运行概况反映软件在实际过程中将是如何使用的。它包含对输入类型的描述以及这些输入发生的可能性。当一个新的软件系统替换已存在的手工作业方式或自动化系统时, 很自然地就能得到新软件可能的使用模式。新软件的使用模式应该大致上和现存的使用操作模式相仿, 只是需要根据新软件所提供的功能进行适当调整。举例来说, 对于一个电信交换系统可以给出它的运行概况, 因为通信公司知道这个系统所需要处理的调用模式。

运行概况中关于输入及其可能性的典型描述如图 15-4 所示。那些发生概率极高的少数输入类型列于图中的最左侧, 还有大量输入类型, 它们发生的可能性是相当低的, 但又不是不可能发生的, 这些类型输入列于图中的右侧, 图中省略号表示该图并未详尽列出所有的输入类型。

Musa (1998) 讨论了他所从事的电信领域中运行概况的发展。因为收集用法数据已经有很长的历史了, 所以运行概况开发过程相对比较简单。文档仅仅反映了历史的用法数据。对于一个需要大约 15 人年方能开发完成的系统, 运行概况大约需要 1 人月就可以完成。而在其他情形中, 运行概况生成要长得多 (2~3 人年), 当然, 这个成本最后分解到了系统多个发布版本中了。Musa 经过对他所在的公司 (一家电信公司) 的统计发现, 在开发运行概况上的投资至少有 10 倍的回报。

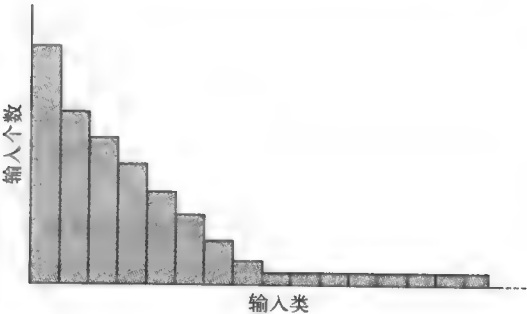


图 15-4 运行概况

不过, 当一个软件系统是新的或者有所创新的时候, 预期它将如何使用是很困难的。因此, 得到准确的运行概况是不太可能的。系统的用户对系统有不同的期待, 他们的背景不尽相同, 而且他们也会有不同的经验, 没有可以沿袭的用法数据库。计算机用户对系统的使用方式往往超出设计者的估计。

对于一些像电信系统这样使用一个标准化的模式的系统来说, 开发准确的运行概况是完全有可能的。然而, 对于其他类型的系统来说, 有很多不同的用户, 他们以各自不同的方式使用系统。正如第10章中所讨论的, 因为使用系统的方式不同, 不同的用户可能得到截然不同的系统可依赖性印象。

而且问题不仅如此, 运行概况不是静止不变的, 而是会随着系统的使用不断变化。当用户使用系统的经验不断增长时, 用户的能力和信心也在不断增加, 他们会用更复杂的方式使用系统。因为这些方面的原因, 开发出一个准确的运行概况有时几乎是不可能的。因此, 不能对任何可依赖性度量的准确性有十足的把握, 因为它们可能是基于系统使用方式的不准确的假设。

15.3 信息安全测试

当越来越多的系统连接到互联网时, 系统的信息安全评估开始逐渐地重要起来。基于网络的系统常常受到攻击, 病毒和蠕虫使用因特网协议到处散布。

所有这些意味着基于网络系统的检验和有效性验证必须集中在信息安全评估上。信息安全评估要测试系统抵御各种类型的攻击的能力。但是, 正如 Anderson (Anderson, 2001) 所说, 这种信息安全评估很难开展。结果, 系统总存在可使攻击者获得存取权或摧毁系统或破坏数据的安全漏洞。

从根本上讲, 信息安全很难评估的原因有两个:

1. 在于信息安全性需求, 就像某些安全性需求一样, 是“不应该”的需求。这就是说, 它们定义了哪些系统行为是不允许发生的, 而不是定义了期待发生的行为。然而, 并不总是能用一些系统容易检查的简单约束来定义这些行为。

如果资源可用 (至少在原理上), 你总能证明系统满足功能需求。但是很难证明系统不能做某事。所以系统不管有多少的测试, 信息安全的脆弱性仍能保留在系统中。当然, 你可以生成一些功能性的需求, 设计它们用来保护系统免遭一些已知类型的攻击。但是, 你不能由未知的和不能预期类型的攻击派生出相应的需求。即使对一个久经考验的系统, 机灵的攻击者也能发现新的攻击方式, 从而可以进入看似安全的系统中。

2. 攻击系统的人都很聪明, 并且很积极地去发现系统的弱点。他们很乐意拿系统做实验, 并且尝试那些非同寻常的行为和系统使用。例如, 在一个姓氏域内, 他们可能输入1000个混合着字母、标点和数字的字符。此外, 一旦他们发现一个漏洞, 他们会交换关于漏洞的相关信息, 来增加可能攻击者的数目。

攻击者可能尝试发现系统开发者所做出的假设, 然后做出违反这些假设的动作以观察出现怎样的结果。他们处在一个利用和探索系统的时期, 并且使用软件工具来分析系统发现可以被他们利用的漏洞。实际上, 他们可能比系统测试工程师花更多的时间来查找漏洞, 作为测试者必须也将焦点集中在测试系统之上。

基于这样的原因, 静态分析能够成为一个特别有用的系统测试工具。一个程序的静态测试可以很快地指引测试团队找到可能包含错误和漏洞的地方。静态分析中发现的异常能够直接被修订或者能够帮助鉴别测试是否这些异常代表着系统存在风险。

检查系统的信息安全性, 可以使用组合测试, 基于工具的分析 and 形式化的检验:

1. **基于经验的测试** 在这种情形中, 检验小组根据掌握的攻击类型对系统进行分析。这可

能包括开发测试用例或者检查系统源代码。例如，检查不为众人所周知的 SQL 语句攻击的系统，可以使用包含 SQL 语句的命令测试系统。检查缓冲区溢出错误是否会发生时，可以检查所有输入缓冲区，看是否程序中缓冲区元素的赋值都是在界内的。

这种类型的有效性验证通常与基于工具的有效性验证一起进行，工具能够提供信息并且帮助把重点放在系统测试上。创建信息安全问题的检查表有助于有效性验证过程。图 15-5 给出了可能驱动基于经验测试的问题的一些例子。对于信息安全性设计和编程指引（第 14 章）是否被遵循的检查也可能被包含在一个信息安全性问题检查表中。

2. 老虎小组 这是一种基于经验的测试形式，可以从开发团队外部利用经验来测试应用系统。专门建立一个老虎小组，其使命就是攻破系统信息安全防线。他们模拟黑客对系统攻击，发挥他们的才智来发现系统漏洞。老虎小组成员应当具有原先安全测试工作的经验，并且找到过系统信息安全性的弱点。

3. 基于工具的测试 对于这种方法，使用各种不同的信息安全工具，如口令检验器分析系统。口令检验器检查不安全的口令，如姓名或由连续字母组成的字符串。这实际上是基于经验的验证的扩展，系统信息安全缺陷的经验被固化在工具中了。当然，静态分析是另一种基于工具的测试方法

4. 形式化检验 对系统按照形式化信息安全描述进行检验。不过，如在其他领域中，形式化检验没有被广泛采用。

信息安全检查表
1. 应用中所有创建的文件都具有适当的访问许可吗？错误的访问许可可能导致哪些被访问的文件被未授权的用户使用
2. 系统是否在用户有一段时间被打断的情况下自动终止用户的此次访问吗？用户的访问过程如果依然有效那会带来通过无看护的计算机进行的未授权的访问
3. 如果系统所用的编程语言是一种没有数组边界检查的语言，是否存在缓存溢出会被利用的情形。缓存溢出会允许攻击者发送代码串到系统并执行它们
4. 如果口令已设定，是否系统检查口令是“强壮”的？强壮的口令包含混合的字母、数字和标点符号，且不会是能在词典里面找到的。它们会比简单口令更难以攻破
5. 来自系统环境的输入总是对照着输入描述进行了检查吗？对坏的格式输入的非正确的处理是导致信息安全脆弱性的一般原因

图 15-5 信息安全检查表的例子

信息安全性测试不可避免地受到测试团队可用的时间和资源的限制。这就意味着你应当采用一个冒险的方法来进行信息安全性测试，并且把注意力集中在你认为的系统最可能出现问题的风险上。如果你对系统信息安全性风险有一个分析，这些分析可以被用来驱动测试的过程。在对这些风险所导出的信息安全需求进行系统测试之外，测试团队也应该试图通过采用另外的威胁系统资产的途径来攻击系统。

对于最终用户来说，检验系统的信息安全是非常困难的。因此，在北美和欧洲政府团体已经建立起了信息安全的评估准则，可以供专业的评估人员使用（Pfleeger 和 Pfleeger, 2007）。软件产品提供商可以提交他们的产品接受基于这些准则的评估和认证。因此，如果你对信息安全需求有特别的要求，你可以选择一种已经得到过相应级别信息安全检验的产品。然而，实际上，这些准则主要还是在军事系统中使用，并没有得到很多商业上的接受。

15.4 过程保证

正如 13 章中所讨论的，经验表明可靠的过程产生可靠的系统。这就是说，如果一个过程基

于好的软件工程实践,那么更有可能产生可靠的软件产品。当然,一个好的过程并不能保证完全的可依赖性。但是,证据表明人们使用了可靠的过程来提升整体的系统可依赖性的信心。过程保证是关于系统开发中所使用的信息以及这些过程的结果的收集的活动。这些信息为软件开发期间所执行的分析、复查和测试工作提供证据。

过程保证关乎两个方面的事情:

1. 我们是否拥有正确的过程? 是否在机构里使用了系统开发过程, 包括对于被开发系统类型的适当控制和 V&V 子过程?
2. 我们正在进行的过程是否正确? 机构是否如它的软件过程描述所定义的那样执行它的开发工作的, 并且是否已经明确规定了软件过程的结果 (输出)?



软件的管理

管理者是由政府指定的来保证私人企业不会通过躲避国家有关安全 and 信息安全等标准的方式获利。很多像原子能、航空和银行业这样的产业中都有管理部门。在要求极高的国家基础设施建设中软件系统变得越来越重要, 这些管理部门也逐渐对软件系统的信息安全性和可依赖性用例更加关注。

<http://www.SoftwareEngineering-9.com/Web/DepSecAssur/Regulation.html>

对于那些在要求极高系统工程方面有大量经验的公司来说, 它们已经摸索出自己的能反映良好的检验和有效性验证的实践。在有些时候, 这还包括它们与外部管理部门在使用什么样的过程方面达成的一致意见。尽管在公司之间有很多过程上的不同, 但是在要求极高的系统开发过程中, 应当包含的活动有: 需求管理, 变更管理, 配置控制, 系统建模, 复查以及审查, 测试计划, 测试覆盖分析。过程改善的概念, 即在过程中引入好的实践经验并制度化, 将在第 26 章讨论到。

过程保证的另一个方面是检查是否过程得到了正确的贯彻执行。这通常包括确保过程产生正确的文档, 并检查过程的文件材料。例如, 一个可依赖过程的一部分会包括形式化的程序审查。每次审查所涉及的文档应当包括用来驱动审查的检查表、人员列表, 以及审查过程中发现的问题和要求采取的行动。

因此, 要证明使用了一个可依赖的过程应包括生成一大批关于开发过程的文档类材料和软件本身。对于这样的大量文档的需要, 意味着敏捷过程在要求信息安全性和可依赖性认证的系统中很少使用。敏捷过程侧重于软件本身, 以及 (坚定地) 主张大量的过程文档从生成以后再也没有被使用过。然而, 当过程信息作为系统安全性或可依赖性用例的一部分时, 就必须创建证据和记载过程的活动。



软件工程师许可

在许多应用类型中, 负责安全的系统工程师必须是经过认证的。无经验的、资质差的工程师是不能对安全负责的。目前对软件工程师人员还没有这样的规定。虽然美国某些州颁发了软件工程师执照 (Knight 和 Leveson, 2002; Bagert, 2002)。然而, 将来对信息安全性要求极高的软件开发的过程标准可能需要的安全工程师应该是正式认证的, 具备某一最低水平的资格和经验。

<http://www.SoftwareEngineering-9.com/Web/DepSecAssur/Licensing.html>

安全保证过程

大部分过程保证的工作已经用于安全性要求极高的系统开发领域中。在安全性要求极高的系统开发过程中包含 V&V 过程是很重要的，V&V 过程用于安全性分析和保证。有两个原因：

1. 事故在要求极高系统中是罕见的事件，不可能在测试过程中实际地模拟它们。不能指望用全面测试重复事故发生的条件。

2. 如第 12 章中讨论的，安全性需求有时是“不应该”类型的需求，它可以排除不安全系统行为。它是不可能通过测试和其他检验需求得到满足的有效性验证活动表现出来的。

专门的安全保证活动一定要包含在软件开发过程当中的所有阶段当中。这些安全性保证活动记录所进行的分析以及负责这些分析的人员。被加入到软件过程中的安全性保证活动可能包括以下这些方面：

1. 危险日志和监控，从初步危险分析到测试再到系统有效性验证全过程跟踪危险。

2. 安全复查，贯穿在整个开发过程中的安全性复查。

3. 安全认证，对安全性要求极高的组件进行正式的安全认证。由一个独立于系统开发团队的外部人员组成的小组检查相关的证据，并且在它投入使用前决定一个系统或组件是否是安全的。

为了支持这些安全性保证过程，应该指定项目安全工程师，他应当对系统的安全性方面负有明确的责任。这就意味着这些人在系统安全性故障出现时应该承担责任。他们必须能够证明安全性保证相关步骤已经得到了正确的执行。

安全工程师和质量经理一起工作，确保使用一个详细的配置管理系统追踪所有安全相关文档，并且保持系统和相关的技术文档同步。对于所有的可靠系统这都很关键。如果一个配置管理故障意味着将错误的系统交付给了客户，那么严格的有效性验证过程也就没什么意义了。配置和质量管理的将在第 24 章和第 25 章中讲述。

危险分析过程作为安全性要求极高的系统开发过程中一个重要部分，它是安全性保证过程的一个例子。危险分析重点在识别危险及其发生概率。危险概率指的是导致事故的危险产生的概率。如果有程序代码检查并处理每个危险，就可以证明这些危险不会造成事故了。这样的论证可以辅之以在本章后面的章节中会讨论到的安全性论证。在一个系统使用前需要得到外部认证的时候（例如，飞机中的软件系统），能证明软件具有跟踪能力通常是认证的前提条件。

应该生成的核心安全文档是危险日志。这个文档提供了有关如何识别在软件开发过程中所考虑的危险的证据。这个危险日志要用于软件开发过程的每个阶段，记录在各个开发阶段是如何考虑这些危险的。胰岛素泵系统的一个简化的危险日志条目如图 15-6 所示。它记录了危险分析的过程并显示了在这个过程中生成的设计需求。该设计需求目的是确保控制系统不会产生过量的胰岛素。

如图 15-6 所示，每个安全负责人都要在条目中明确地标明，这是很重要的，因为有以下两方面的原因：

1. 当人员被确定时，他们可以对自己的行动负责。这就意味着他们很可能更关心自己的工作，因为任何问题都可以追踪到他们的工作上。

2. 在发生事故时，很可能产生法律诉讼或调查。能够确认对安全保证的负责人很重要，这样他们就能够说明他们的行为。

危险日志		第 4 页：打印日期 2009 -02 -20		
系统：胰岛素泵系统		文件：InsulinPump/Safety/HazardLog		
安全工程师：James Brown	日志版本：1/3			
识别出的风险	传输了过量的胰岛素给病人			
识别人	Jane Williams			
危险类型	1			
识别出的风险	高			
缺陷树识别	是	日期	24.01.07	位置 危险日志 第 5 页
缺陷树创建人	Jane Williams 和 BillSmith			
缺陷树检查	是	日期	2007 -01 -28	检查人 James Brown
系统安全性设计需求： 1. 系统必须包含自检测软件，能够测试传感器系统、时钟以及胰岛素传输系统。 2. 自检测软件必须每一分钟执行一次。 3. 当自检测软件无论在哪个系统组件中发现了缺陷，必须能发出声音报警，泵显示器必须指示缺陷所在的组件的名字，胰岛素传输因而要暂停。 4. 系统需要插入一个干预系统允许系统用户修改通过计算得到的胰岛素剂量。 5. 干预的量一定不能大于预设值（最大干预量），这是医务人员对系统事先设置好的。				

图 15-6 简化了的危险日志条目

15.5 安全性和可依赖性案例

信息安全性和可依赖性保证过程会产生大量的信息。其中可能包括测试结果，使用的开发过程信息，复审会议记录等信息。这样的信息提供了系统的信息安全性和可依赖性证据，并且帮助判断系统是否对于运行使用有足够的可依赖性。

安全案例和可依赖性案例，是一种结构化文档，用来给出有关系统是安全的或达到了所需要的可依赖性的水平的详细论证和证据。它们有时被称为保证案例。根本上说，一个安全性或者可依赖性案例就是把所有可用的证据放在一起，展示系统是值得信任的。对于很多要求极高的系统，具有安全案例的产品是法律上的需要，并且在系统使用前案例必须满足某种认证要求。

管理部门的职责是检查一个完全的系统的实用性、安全性和信息安全性。所以当开发项目完成时，外部管理部门一般就要介入。但是管理部门和开发者几乎不是孤立工作的；他们与开发小组进行交流以确定哪些东西应该包含在安全案例当中。管理部门和开发者联合起来检查过程和步骤以保证这些过程和步骤都能按照管理部门的要求实施和文档化。

可依赖性案例通常在系统开发过程中或之后开发。在开发过程活动不能生成系统可依赖性的证据的情况下，有时可能会导致问题的发生。Graydon 等（2007）认为安全性和可依赖性案例的开发应该紧紧和系统设计和实现结合在一起。这就意味着系统设计的决策会受到可依赖性案例的需求的影响。那些可能明显增加案例开发的难度和花费的设计选择必须尽量避免。

可依赖性案例是系统安全性案例的一般化。安全案例是一组文档，它包括经过确认的系统描述、开发系统所使用的过程的相关信息，以及更重要的能证明系统是安全的逻辑论证。Bishop 和 Bloomfield（1998）给安全案例下了个更简洁的定义：

是一组文档化的证据，它提供了令人信服的和有效的论证，证明在给定的环境下系统对于特定的应用是安全的。

安全性和可依赖性案例的组成和内容依赖于所要确认的系统的类型和它的操作上下文。图

15-7 给出了一个软件安全案例的一个可能的组成，但是还没有广泛使用的工业标准。安全性案例的结构根据行业和领域的成熟度各不相同。例如，原子能安全案例已经使用很多年了。它们非常广泛并且以一种原子能工程师所熟悉的方式呈现。但是，医疗设备的安全性案例最近才被引入。它们的结构更加灵活，并且案例本身也没有原子能案例具体。

章 节	描 述
系统描述	系统概述和对其关键组件的描述
安全性需求	从系统需求描述中导出的安全性需求。系统相关需求的详细内容也会包括在内
危险和风险分析	描述所识别的危险和风险以及所采取的降低风险措施的文档。危险分析和危险日志
设计分析	一组结构化的论证（参见 15.5.1 节），以证明设计是安全的
检验和有效性验证	描述所使用的 V&V 过程，适当的地方，还会包括对系统的测试计划。测试结果的总结会给出所检测到的且改正了的缺陷。如果使用了形式化方法，会有一个形式化系统描述和对此描述的其他分析。对源代码的静态分析记录
复查报告	对所有设计和安全复查的记录
团队能力	投入到安全相关的系统开发和验证的所有团队成员的能力的证据
QA 过程	在系统开发过程中所执行的质量保证过程（参见第 24 章）的记录
变更管理过程	对所提出的所有变更、所执行的活动的记录，有的时候，还包括这些变更安全性的判断
相关的安全案例	指向其他会对此安全案例产生影响的安全案例的指针

图 15-7 软件安全案例内容

当然软件本身并不危险。仅仅当它被固化到一个大型的基于计算机或社会技术系统时，软件的失败会导致其他设备或系统的失败从而造成伤亡。所以一个软件安全案例总是更广泛的系统安全案例（展示整个系统的安全）的一部分。当构造一个软件安全案例时，你需要把软件失败同更广泛的系统失败关联起来，说明不会出现这些软件失败，或者是软件失败不会以一种能最终导致系统失败的方式传播。

15.5.1 结构化论证

决定一个系统是否在使用中足够可依赖，这是应该基于逻辑论证的。这些论证应该表明所提出的证据能够支持系统信息安全性和可依赖性的断言。这些断言可能是绝对的（事件 X 将会发生或将不会发生），或可能的（事件 Y 发生的概率是 0. n）。论证就是把证据和断言连接起来。如图 15-8 所示，论证是一个关系，是一种什么将会发生（断言）和一组收集到的证据之间的关系。论证，主要是解释为什么断言能够从得到的证据中推断出来。断言是关于系统信息安全性和可依赖性的断言。

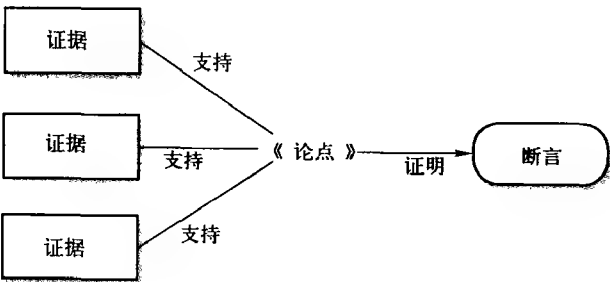


图 15-8 结构化论证

例如，胰岛素泵是一个对安全性要求极高的设备，一旦出现故障就可能伤害到用户。在许多国家里，这就意味着管理部门（在英国是医疗设备理事会）必须在设备售出和使用之前，使系统的安全性让人信服。为了做出这样的决策，管理部门要评估系统安全性案例，这样的案例给出关于系统正常操作不会对用户造成伤害的结构化论证。

安全性案例总是依赖于结构化的基于断言的论证。例如，下面的论证会被用来证明由控制软件所执行的计算将不会导致给予用户过量的胰岛素。当然，这是一个简化了的论证。一个真正的安全案例中应该给出对证据的详细引用。

断言：胰岛素泵的一次最大量不超过 maxDose ，这里 maxDose 是经过核定的对一个特殊病人一次使用的安全剂量。

证据：胰岛素泵软件控制程序的安全论证（在之后的章节中会讨论到安全性论证）。

证据：胰岛素泵的测试数据集。在 400 份测试中， currentDose 的值得到准确计算并且从来没有超过 maxDose 。

证据：胰岛素泵控制程序的静态分析报告。这种控制软件的静态分析呈现没有任何异常影响 currentDose 的赋值，呈现那个表示要注射的胰岛素剂量的程序变量。

论点：给出的安全证据表明可以计算出的胰岛素剂量最大量等于 maxDose 。

总之，有充分的信心去合理地假设，证据证明了这样的一个断言，即胰岛素泵不会计算出一个超出最大单个剂量的传输剂量。

注意到证据的表示既冗余又是多样的。软件经过了多个不同的机制的检查，这些机制之间存在很大的重叠性。如在第 13 章中讨论过，使用冗余性和多样性的过程增加了信心。如果遗漏和错误不能被一个有效性验证过程检查出来，则很可能会被另一个验证过程发现。

当然，通常会有很多关于系统可依赖性和安全性的断言。一个断言的正确性通常取决于其他断言是否是正确的。因此，断言可能被组织成层次的结构。图 15-9 表明了对于胰岛素泵断言层次结构的一部分。为了证明高层断言是正确的，你必须先通过对低层断言的论证。如果你可以说明每一低层的断言都是正确的，那么你就可能能够推断出较高层次的断言也是正确的。

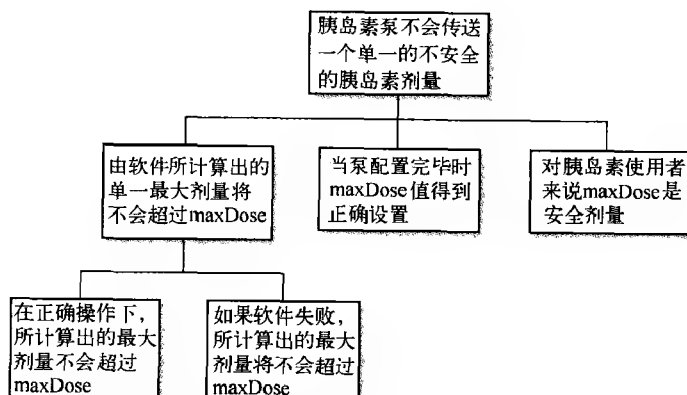


图 15-9 一个胰岛素泵的安全性断言层次结构

15.5.2 结构化的安全性论证

结构化的安全性论证是一种结构化的论证，说明一个程序达到了它的安全性责任。在安全性论证中，并不需要证明程序完全符合系统描述，只需要证明程序的执行不会进入不安全的状

态。这就意味着，安全性论证比正确性论证花费更少。你不需要去考虑所有的程序状态，可以仅仅将注意力集中在可能导致故障的状态上。

一个支持系统安全性方面的工作的一般假设是，能导致安全性要求极高系统危险的系统错误数目远远小于系统中可能存在的总的错误数目。安全性保证可以集中于这些有潜在危险的错误。如果它能够证明这些错误不可能发生，或者是如果发生了，相关的危险也不会导致系统故障，那么这个系统就是安全的。这是结构化的安全性论证的基础。

结构化的安全性论证试图证明，假设运行环境是正常的，那么程序应当是安全的。安全性论证通常是基于矛盾的。在创建安全性论证过程中包括以下几个步骤：

1. 首先假设通过危害分析找出来的一个不安全状态能通过程序执行而到达。
2. 写一个谓词（逻辑表达式）来定义此不安全状态。
3. 然后系统地分析系统模型或程序代码，并给出所有能到达此状态的程序路径的所有终止条件，证明这些路径的终止条件与不安全状态谓词是相矛盾的。如果这样，对不安全状态的初始假设就是不正确的。
4. 如果这个证明过程对所有识别出来的危险都逐个使用过，那就可以证明软件是安全的。

结构化的安全性论证可以应用在不同的水平上，从需求到设计模型再到程序代码。在需求水平上，我们试图证明没有遗漏安全性需求，并且需求没有对系统做出无效的假设。在设计水平上，我们可能去分析一个系统的状态模型以找出不安全状态。在程序代码水平上，我们通过安全性要求极高的代码来考虑所有的路径，以表明会导致矛盾的出现所有执行路径。

作为一个例子，我们来看图 15-10 中给出的胰岛素注射系统中的一段程序代码。这段代码计算出要注射的胰岛素的剂量，然后采用一些安全性检查来降低过量胰岛素被注射的可能性。对这段代码所设计的安全性论证需要说明胰岛素的输入剂量不会超过单个剂量的上限水平。这是通过每个糖尿病患者和他们的医疗顾问相互讨论确定下来的。

```
- The insulin dose to be delivered is a function of
- blood sugar level, the previous dose delivered and
- the time of delivery of the previous dose
currentDose = computeInsulin ();
// Safety check-adjust currentDose if necessary.
// if statement 1
if (previousDose == 0)
{
    if (currentDose > maxDose/2)
        currentDose = maxDose/2 ;
}
else
    if (currentDose > (previousDose * 2) )
        currentDose = previousDose * 2 ;
// if statement 2
if ( currentDose < minimumDose )
    currentDose = 0 ;
else if ( currentDose > maxDose )
    currentDose = maxDose ;
administerInsulin (currentDose) ;
```

图 15-10 有安全性检查的胰岛素剂量计算

为了证明安全性，不必证明系统注射了正确剂量，只要它没有给病人注射过量就可以了。你的工作是建立在假设 maxDose 对于系统用户来说是个安全水平的基础上的。

为了构造安全性论证，要找出一个定义不安全状态的谓词，这里是 $\text{currentDose} > \text{maxDose}$ 。接下来就是要证明所有的程序路径都与这个不安全断言相矛盾。如果是这样，这个不安全条件就不可能是真实的。如果能这样执行，就可以很有信心地说程序不会计算出危险的胰岛素剂量。你可以以图形的形式构造和表示安全性论证，如图 15-11 所示。

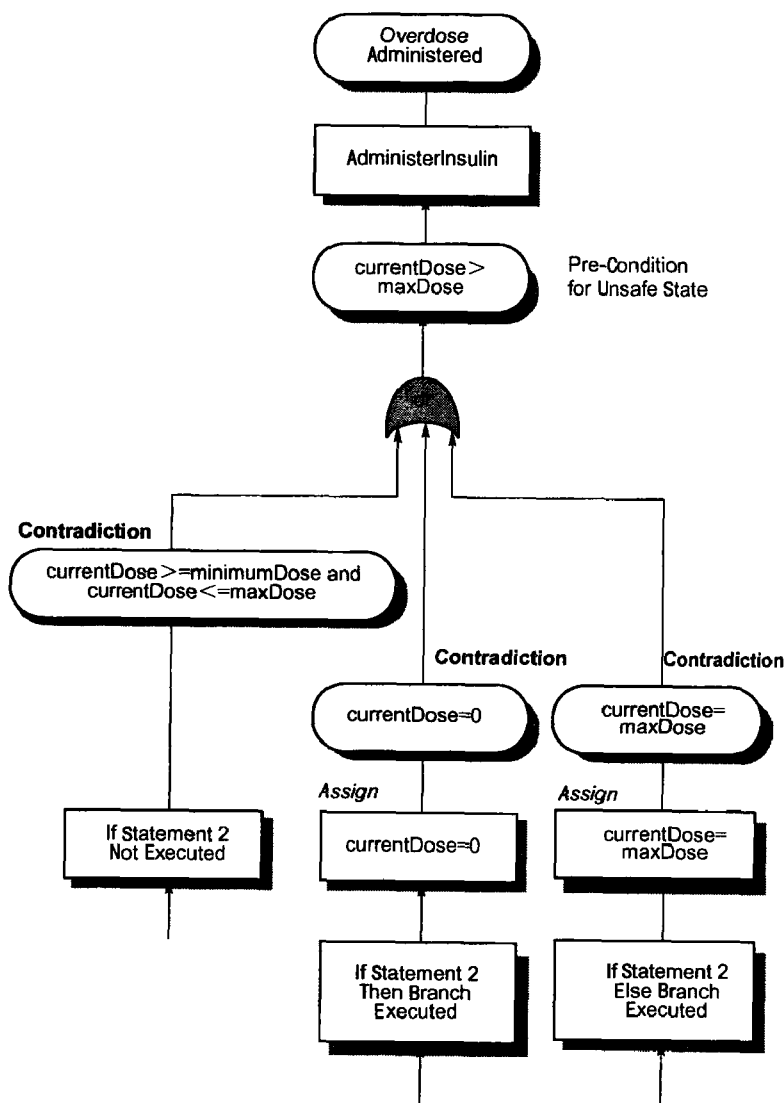


图 15-11 基于矛盾显示的非形式化的安全性论证

为了构造一个结构化的论证，使一个程序不会做出不安全的计算，首先要通过代码确定能够通向潜在的不安全状态的所有可能路径。然后由这些不安全状态出发，反向地工作，考虑通向这个不安全状态的每个路径的所有状态变量的最后一次赋值。如果能够说明这些变量中没有有一个赋值是不安全的，那么就表示你最初的假设（计算是不安全的）是不正确的。

反向工作很重要，因为它表示你能够忽略所有的其他状态，只考虑通向代码出口条件的那些最终状态。之前的值对于系统的安全性已经无关紧要。在这个例子中，所有需要考虑的是在 `administerInsulin` 方法执行之前的对 `currentDose` 的最近一次可能的赋值。在安全性论证中你可以忽略一些计算，如图 15-10 中的 if 语句 1，因为它们的结果都会被之后的语句所覆盖。

图 15-11 中所示的这个安全性论证中，有 3 个可能的程序路径可以引起 `administerInsulin` 方法调用。我们必须说明所传输的胰岛素的数量不会超过 `maxDose`。所有到达 `administerInsulin` 的程序路径包括：

1. if 语句 2 的两个分支都没有执行，这个情况只能发生在当 `currentDose` 大于或等于 `minimumDose` 和小于或等于 `maxDose` 时。这是后置条件，即在语句被执行过后是正确的断言。
2. if 语句 2 的 then 分支被执行。这时对 `currentDose` 赋值 0 的语句执行。因此，它的花置条件是 `currentDose = 0`。
3. if 语句 2 的 else - if 分支被执行，这时对 `currentDose` 赋值 `maxDose` 的赋值语句得到执行。因此，在这条语句被执行后，我们知道花置条件是 `currentDose = maxDose`。

在所有的情况中，花置条件都与不安全状态的花置条件（控制的数量大于 `maxDose`）是矛盾的。因此我们能够说计算是安全的。

结构化的论证在证明系统某种信息安全性特性的正确性时，也能够以同样的方式使用。例如，如果你希望说明一个计算不可能允许对正在修改中的资源的访问，你可以使用一个结构化的安全性论证来说明。但是，来自结构化论证的证据对于信息安全性验证来说不那么可靠。这是因为存在攻击者损坏系统代码的可能性。在这种情况下，执行的代码已经不是你之前所声明的安全代码了。

要点

- 静态分析是 V&V 的一种检查系统源代码（或其他表示）的方法，查找错误和异常。它允许程序的所有部分得到检查，不仅仅是那些会被测试到的部分。
- 模型检测是静态分析的一种形式化方法，它彻底检查系统所有状态以发现潜在的错误。
- 统计性测试用来估计软件可依赖性。它依赖于用测试数据集对系统的测试，这个测试数据集反映软件的运行概况。测试数据有时是可以自动生成的。
- 信息安全验证是有困难的，因为信息安全需求描述在系统中不应该出现什么，而不是应该出现什么。而且，系统攻击者都很聪明，并且可能比安全测试人员有更多的时间来探测系统弱点。
- 信息安全验证可以使用基于经验的分析和基于工具的分析，或者采用“老虎小组”来模拟攻击系统。
- 有一个定义完好的并经过认证了的开发过程对安全性要求极高的系统开发非常重要。这样的过程一定要包括对潜在危险的查找和监控。
- 安全和可依赖性案例搜集所有能表明系统是安全和可靠的证据。当外部管理部门在系统使用前对系统进行验证时需要用到这些安全案例。
- 安全性案例总是基于结构化的论证。结构化的安全性论证说明一个识别出的危险条件永远不会发生，通过考虑所有可能到达不安全条件的程序路径，说明这个条件不能成立。

进一步阅读材料

《Software Reliability Engineering: More Reliable Software, Faster and Cheaper, 2nd edition》, 这是一本使用运行概况和可依赖性模型来评估可依赖性的权威著作。书中包括统计测试使用的经验详细介绍 (J. D. Musa, McGraw-Hill, 2004)。

《NASA's Mission Reliable》讨论 NASA 怎样使用静态分析和模型检测来确保飞船软件的可依赖性 (P. Regan and S. Hamilton, IEEE Computer, 37 (1), January 2004)。http://dx.doi.org/10.1109/MC.2004.1260727。

《Dependability cases》一个基于例子的介绍文章, 定义可依赖性用例的 (C. B. Weinstock, J. B. Goodenough, J. J. Hudak, Software Engineering Institute, CMU/SEI-2004-TN-016, 2004)。http://www.sei.cmu.edu/publications/documents/04.reports/04tn016.html。

《How to Break Web Software: Functional and Security Testing of Web Applications and Web Services》一本简短的提供关于怎样对网络应用运行安全测试的良好实践建议的一本书 (M. Andrews and J. A. Whittakeer, Addison-Wesley, 2006)。

《Using static analysis to find bugs》这篇文章介绍了“FindBugs”, 一个使用简单技术来查找潜在信息安全侵害和运行时错误的 Java 静态分析器。(N. Ayewah et al., IEEE Software, 25 (5), Sept/Oct 2008)。http://dx.doi.org/10.1109/MS.2008.130。

练习

- 15.1 解释一下在要求极高软件系统开发中何时使用形式化描述和检验是划算的。以及你为什么认为要求极高系统的工程师会不赞成使用形式化方法?
- 15.2 列出一个条件表, 其中的条件是能够被 Java、C++ 或是其他你正在使用的编程语言的一个静态分析器所发现。对比图 15-2 给出的表, 对这个条件表做出评论。
- 15.3 如果可依赖性描述表示为系统整个生命周期中的非常少量失败, 解释为什么验证这样的可依赖性实际上是不可行的。
- 15.4 解释为什么保证系统可依赖性不能确保系统的信息安全性。
- 15.5 使用例子解释为什么信息安全性测试是一项困难的过程。
- 15.6 试说明你将如何验证所开发的应用中的口令保护系统, 解释任何你认为有用的工具的功能。
- 15.7 MHC-PMS 应对可能透露机密的病人信息的攻击时必须保证安全。这样的攻击在第 14 章中讨论过。利用这些信息, 对图 15-5 的检查表进行扩展, 用来对 MHC-PMS 的测试者给予指导。
- 15.8 列出需要系统软件安全案例的 4 种系统类型。解释为什么需要安全案例。
- 15.9 核废料储存设施中的门锁控制机制是为了安全操作。它能保证只有当在辐射防护罩保护的情况下或辐射水平降低到一给定值 (危险水平) 的情况下方可进入, 这就是说:
 - (i) 远程控制的辐射防护罩安装在房间里面, 门可以由经认可的操作人员开启。
 - (ii) 如果房间里的辐射水平低于给定数值, 门可以由经认可的操作人员开启。
 - (iii) 经认可的操作人员是通过输入授权的入口密码确认的。
 在图 15-12 中给出的代码是用来控制门锁机制的程序段。注意这里安全状态是入口不应该准进入。使用在 15.5.2 节中讨论的方法, 给出此代码的一个安全论证。使用行数指定特定的语句。如果你发现代码是不安全的, 对如何修改该代码使之安全给出建议。
- 15.10 假设你是某化工厂软件开发团队中的一员。此软件发生了失败, 导致了严重的污染事故。你的老板接受了电视台的采访, 并声称软件的验证过程是全面详尽的, 在软件中不存在缺陷。她断言问题一定出在不好的操作流程上。有一家报纸希望得到你对事故的观点。讨论你如何处理这样的一次采访。

```

1  entryCode = lock.getEntryCode () ;
2  if (entryCode == lock.authorizedCode)
3  {
4      shieldStatus = Shield.getStatus ();
5      radiationLevel = RadSensor.get ();
6      if (radiationLevel < dangerLevel)
7          state = safe;
8      else
9          state = unsafe;
10     if (shieldStatus == Shield.inPlace() )
11         state = safe;
12     if (state == safe)
13     {
14         Door.locked = false ;
15         Door.unlock ();
16     }
17     else
18     {
19         Door.lock ( );
20         Door.locked := true ;
21     }
22 }

```

图 15-12 门禁代码

参考书目

- Abrial, J. R. (2005). *The B Book: Assigning Programs to Meanings*. Cambridge, UK: Cambridge University Press.
- Anderson, R. (2001). *Security Engineering: A Guide to Building Dependable Distributed Systems*. Chichester, UK: John Wiley & Sons.
- Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. Cambridge, Mass.: MIT Press.
- Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., S. K., R. and Ustuner, A. (2006). 'Thorough Static Analysis of Device Drivers'. *Proc. EuroSys 2006*, Leuven, Belgium.
- Bishop, P. and Bloomfield, R. E. (1998). 'A methodology for safety case development'. *Proc. Safety-critical Systems Symposium*, Birmingham, UK: Springer.
- Chandra, S., Godefroid, P. and Palm, C. (2002). 'Software model checking in practice: An industrial case study'. *Proc. 24th Int. Conf. on Software Eng. (ICSE 2002)*, Orland, Fla.: IEEE Computer Society, 431-41.
- Croxford, M. and Sutton, J. (2006). 'Breaking Through the V and V Bottleneck'. *Proc. 2nd Int. Eurospace—Ada-Europe Symposium on Ada in Europe*, Frankfurt, Germany: Springer-LNCS, 344-54.
- Evans, D. and Larochelle, D. (2002). 'Improving Security Using Extensible Lightweight Static Analysis'. *IEEE Software*, 19 (1), 42-51.
- Graydon, P. J., Knight, J. C. and Strunk, E. A. (2007). 'Assurance Based Development of Critical Systems'. *Proc. 37th Annual IEEE Conf. on Dependable Systems and Networks*, Edinburgh, Scotland: 347-57.
- Holzmann, G. J. (2003). *The SPIN Model Checker*. Boston: Addison-Wesley.

Knight, J. C. and Leveson, N. G. (2002). 'Should software engineers be licensed?' *Comm. ACM*, **45** (11), 87–90.

Larus, J. R., Ball, T., Das, M., Deline, R., Fahndrich, M., Pincus, J., Rajamani, S. K. and Venkatapathy, R. (2003). 'Righting Software'. *IEEE Software*, **21** (3), 92–100.

Musa, J. D. (1998). *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*. New York: McGraw-Hill.

Nguyen, T. and Ourghanlian, A. (2003). 'Dependability assessment of safety-critical system software by static analysis methods'. *Proc. IEEE Conf. on Dependable Systems and Networks (DSN' 2003)*, San Francisco, Calif.: IEEE Computer Society, 75–9.

Pfleeger, C. P. and Pfleeger, S. L. (2007). *Security in Computing, 4th edition*. Boston: Addison-Wesley.

Prowell, S. J., Trammell, C. J., Linger, R. C. and Poore, J. H. (1999). *Cleanroom Software Engineering: Technology and Process*. Reading, Mass.: Addison-Wesley.

Regan, P. and Hamilton, S. (2004). 'NASA's Mission Reliable'. *IEEE Computer*, **37** (1), 59–68.

Schneider, S. (1999). *Concurrent and Real-time Systems: The CSP Approach*. Chichester, UK: John Wiley and Sons.

Visser, W., Havelund, K., Brat, G., Park, S. and Lerda, F. (2003). 'Model Checking Programs'. *Automated Software Engineering J.*, **10** (2), 203–32.

Voas, J. (1997). 'Fault Injection for the Masses'. *IEEE Computer*, **30** (12), 129–30.

Wordsworth, J. (1996). *Software Engineering with B*. Wokingham: Addison-Wesley.

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P. and Vouk, M. A. (2006). 'On the value of static analysis for fault detection in software'. *IEEE Trans. on Software Eng.*, **32** (4), 240–5.

高级软件工程

以“高级软件工程”作为本部分的题目，是因为读者必须了解第1章到第9章的基本原理方可深入学习本部分的内容。在这一部分中讨论的很多话题反映的是在开发分布式系统和实时系统过程中的产业软件工程实践。

软件复用目前已经变成一种对于基于 Web 的信息系统和企业系统的最重要的开发范型。最为常用的复用方法是 COTS 复用，即为满足机构的需要通过对大型系统的配置而很少或基本没有任何源代码开发的软件构造。第16章将介绍复用的一般话题，并着重介绍 COTS 系统复用。

第17章仍然是关于软件复用，但主要介绍组件的复用而不是整个系统的复用。基于组件的软件工程是组件组合的过程，通过新代码的开发将可复用的组件集成在一起。在这一章中，解释组件的内涵和外延，解释为什么需要标准的组件模型来有效地进行组件复用，还讨论基于组件的软件工程的一般过程以及组件组合中会出现的问题。

现在绝大多数的大型系统都是分布式系统，第18章介绍构建分布式系统的理论和技术以及存在的问题。本章介绍基础性的分布式系统工程的范型——客户机-服务器方法，解释实现体系结构风格的各种方式。这一章最后一小节讲解这样一个问题，即提供软件作为分布式应用服务将会怎样彻底地改变软件产品的市场。

第19章介绍面向服务的体系结构相关话题，将分布和复用两个重要概念联系起来。服务是可复用的软件组件，它的功能可以通过互联网访问并对广大客户开放。在这一章中，还解释在创建服务（服务工程）和组合服务以创建一个新软件系统过程中所需要的一些活动。

嵌入式系统是软件系统最为广泛使用的实例。第20章将覆盖此重要话题。本章介绍实时嵌入式系统的思想，并描述3个使用在嵌入式系统设计中的体系结构模式，然后解释时序分析过程，给出实时操作系统的讨论并总结本章内容。

最后，第21章讲解面向方面的软件开发（AOSD）。AOSD 也是与复用相关，基于方面（aspect）这个概念，提出一种新的组织和构造软件系统的方法。尽管它今天还远没有成为软件工程的主流技术，AOSD 具有巨大潜力，会在未来对改善我们目前的软件实现方法起到重要的推动作用。

软件复用

目标

本章的目标是介绍软件复用，以及以大规模系统复用为基础的系统开发方法。读完本章，你将了解到以下内容：

- 了解在开发新系统时复用软件的好处以及在复用中可能出现的一些问题；
- 了解应用框架作为一个可复用对象集合的概念，以及在应用开发中如何使用应用框架；
- 介绍由通用内核体系结构以及可配置可复用组件所构成的软件产品线；
- 掌握通过配置和组合商业现货应用软件系统来开发系统的方法。

基于复用的软件工程是一个软件工程策略，使开发过程适应对现存软件的复用。尽管复用作为一种开发策略已经提出 40 多年（McIlroy, 1968），但是只是从 2000 年以来，复用开发才变成新商业系统的规范。这种向基于复用的开发方式的转变是为了降低软件产品和维护的成本，更快地交付系统，以及提高软件质量。越来越多的公司将其软件看成是一种有价值的资产。他们希望提升复用水平来增加他们在软件上投资的回报。

可复用软件方法使开发软件变得越来越容易。开源运动意味着有一个低价提供的巨大的可重复使用的代码库。这些可重复使用的代码可能是以程序库或整个应用程序的形式来复用。有许多特定领域的应用系统通过裁剪来满足一个特定公司的需要。一些大公司为他们的客户提供一系列的复用组件。标准，比如 Web 服务标准，使得在多种应用范围里开发通用服务和再利用它们变得越来越容易了。

以复用为基础的软件工程是一种开发方法论，其目标是最大化地复用现存软件。被重复使用的软件会有截然不同的规模。举例来说：

1. **应用系统复用** 整个系统可以通过不改变地融合到其他系统当中得以复用或是通过为不同的客户配置应用。或者是通过开发具有共同体系统结构的应用族，而对专业客户进行特殊裁剪实现。本章稍后将介绍应用系统复用。

2. **组件复用** 应用系统的组件规模从子系统到单个对象都可以复用。举例来说，作为文本处理系统的一部分的模式匹配系统可以复用到数据库管理系统中。第 17 章和第 19 章将讲述组件复用。

3. **对象和函数复用** 实现一个单一功能的软件组件，例如数学函数，或者是一个对象类，可以重复使用。这种基于标准函数库的复用形式已经广泛使用了 40 年。很多函数库和类库都可以免费获得。你可以通过连接新开发的应用程序代码来使用库中的函数和类。在数学算法和图形这些领域，需要十分专业的知识来开发高效率的对象和函数，对象和函数复用是尤其奏效的方法。

软件系统和组件是很有潜力的可复用实体，但是它们的特殊性质有时意味着为一个新情况修改它们代价是很大的。对它的一个补救形式是所谓的“概念复用”，它不是复用一个软件组件，而是重复使用一个思想、一个方法、一个操作或一个算法。复用的概念代表一个抽象的概念（例如，一个系统模型），其中不包括实现细节。因此针对一系列的情形，可以对它进行配置和调整。概念复用可以嵌入在设计模式、可配置系统产品以及程序生成器等这样的方法中。在复用

概念时，其复用过程包括一个活动，在这个活动中抽象概念被实例化来生成可执行的复用组件。

软件复用的一个明显好处是总体开发成本得到降低。需要定义、设计、实现和有效性验证的软件组件数目减少了。不过，成本缩减只是复用的好处之一。其他一些复用软件的好处如图 16-1 所示。

优 势	解 释
增加可靠性	在运行着的系统中实际使用的组件要比一个新组件的可靠性高。它们可能已经得到了各种环境的实际使用和测试。在最初的组件使用当中组件在设计和实现上的缺陷都已经暴露出来了，并得到了修正，因而组件在复用时的失败数是下降的
降低过程风险	如果组件已经存在，复用组件的成本不确定性较之开发成本的不确定性减少，这是项目管理当中的一项重要因素，因为这降低了项目成本估计当中的不确定性。在有相对来讲大规模的组件（如，子系统复用）时这一点更明显
专家的有效使用	不用专家在不同项目中做重复的工作，而是让他们开发可复用的组件，用这些组件来封装他们的知识
标准一致	有些标准，如用户界面标准，可以实现为一组标准组件。例如，可以将用户界面中的菜单实现为一个可复用组件。所有的应用都以相同的菜单格式呈现在用户面前。使用标准用户界面增强了可靠性，因为用户面对相同的用户界面，出现错误的可能性降低
加速开发	尽快让系统走向市场要比总开发成本更重要。复用组件加速了系统产品的形成，因为无论开发还是有效性验证时间都缩短了

图 16-1 软件复用的优势

然而，复用会产生成本也会出现特殊的问题（见图 16-2）。了解是否组件对于某个特殊情形是适合复用的，以及在对组件进行测试以确保它的可靠性时会产生可观的成本。这些额外的成本意味着通过复用总的开发成本的降低不会像人们所期待的那么多。

问 题	解 释
增加维护成本	如果组件的源代码是不可得的，那么维护成本将要增加，因为随着系统变更，系统中复用成分的不相容性在增加
缺乏工具支持	CASE 工具集没有对复用的支持，将这些工具与组件库系统集成是困难甚至是不可能的。由这些工具承担的软件过程没有将复用考虑在内
孤芳自赏	有些软件人员有时更愿意重新写一个组件，因为他们相信他们能改善可复用组件。一方面是源于信任因素，另一方面重写软件更具挑战性
创建维护和使用一个组件库	填充组件库并保证软件人员能使用这个库是很费钱的。要调整开发过程的保证组件库可用
查找、理解和改编可重用组件	软件组件必须从库中找出来、理解、有时还需要作些调整使之适应当前的环境。软件工程人员对在库中找到组件应该有合理的自信，然后才能将找组件纳入正常开发过程中

图 16-2 复用的问题

正如第 2 章讨论过的，软件开发过程必须调整以适应复用。特别是，必须有一个需求细化阶段，即对该系统的需求进行修改，以反映可以得到的可复用软件。该系统的设计和实现阶段，可能还包括明确的活动，以寻找和评估备用的可复用组件。

当我们有计划地将软件复用作为一个机构范围的复用计划的一部分时，软件复用是最有效的。一个复用计划包括可复用资源的创建，以及把这些资源合并到新的软件中的开发过程的调整。在日本人们认识到复用计划的重要性已经有很多年了（Matsumoto, 1984），复用是日本的软件开发的“工厂”方法的一个重要组成部分（Cusamano, 1989）。像 Hewlett-Packard 这样的公司

也已经在他们的复用计划上非常成功了 (Griss 和 Wosser, 1995), 他们的经验已经被 Jacobsen 等 (Jacobsen 等, 1997) 记录在书中了。

16.1 复用概览

在过去的 20 年间, 出现了很多支持软件复用的技术。这些技术都利用了这样的事实, 即在同一个应用领域中的系统是相似的, 有可复用的潜在可能; 从简单函数到完整应用, 复用可以在不同的层面上进行; 可复用组件的标准将推动复用。图 16-3 提出了实现软件复用的多种可能的方法, 针对每一种方法, 将在 16.4 节中做简要的介绍。

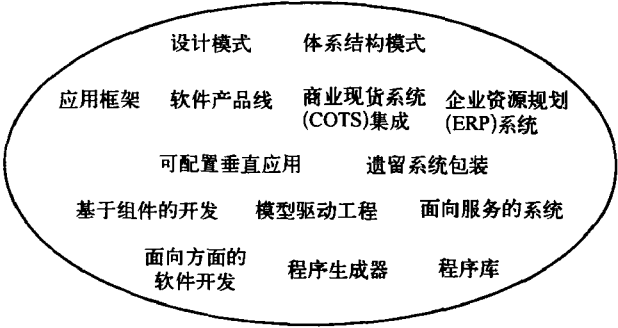


图 16-3 复用概览

有了这一系列的复用技术后, 关键性的问题是“在一个特定的情形中哪种技术是最为恰当的呢?” 显然, 这依赖于所要开发的系统的需求、所拥有的技术和可复用的资产、开发团队的专业知识等。在规划复用时需要考虑的关键因素是:

1. **软件的开发进度表** 如果软件需要快速开发, 你就应该努力复用现成系统而不是对单个组件复用。这是大粒度可复用资产。尽管这可能对于需求来说不是太合适的, 但该方法能最小化开发工作量。
2. **所预计的软件生命期** 如果你所开发的系统是长生命期系统, 你应该主要关注系统的可维护性。你不要仅考虑眼前的复用的好处, 而是要考虑长期影响。在它的整个生命周期中, 你将必须修改系统使之适应新需求, 这意味着需要改变部分系统。如果你不能够得到源代码的话, 那就要避免使用外部供应商所提供的现成的组件和系统, 供应商可能不会持续地对所复用软件给予支持。
3. **开发团队的背景、技术和经验** 所有的复用技术都相当复杂, 需要很多时间来理解和有效地使用。因此, 如果开发团队在某个特别领域有专长, 那这就是我们应该集中注意力的地方。
4. **软件的危险程度和它的非功能性需求** 对于要求极高的一类系统, 它们需要外部管理者的认证, 你可能必须创建系统的可靠性用例 (在第 15 章中介绍过)。如果你不能够得到软件的源代码的话, 那这将是非常困难的。如果你的软件有苛刻的性能要求, 像基于生成器的复用策略就不太可能使用, 在生成器复用中, 你可以从一个可复用的系统领域专门表示中生成代码。这些系统会生成相当低效的代码。
5. **应用领域** 在某些应用领域, 例如制造和医药信息系统, 有多个一般产品, 可以通过对它们重新配置来复用它们使之适应自己的要求。如果在这些领域中工作, 那就应该考虑这些选择。
6. **系统运行的平台** 很多组件模型, 例如, .NET 是 Microsoft 平台专用的。同样地, 一般性的应用系统是平台相关的, 你只能使用那些与你的系统所在平台相一致平台上的组件。

方 法	描 述
体系架构模式	支持普通应用系统类型的标准软件体系结构用作应用程序的基础。详细描述在第6、13、20章
设计模式	将应用间所发生的一般抽象表示为设计模式，这些设计模式给出了抽象和具体的对象和交互。详细描述在第7章
基于组件的开发	通过集成符合组件模型标准的组件（对象集合）来开发系统。详细描述在第17章
应用框架	抽象和具体的类的集合通过调整和扩展来创建应用系统
遗留系统包装	遗留系统（参见第9章）是可以通过包装被复用的，这种包装是通过定义一组接口和提供通过这些接口来访问这些遗留系统的能力来实现的
面向服务的系统	系统通过连接可能来自外部的共享服务而建立。详细描述在第19章
软件产品线	将某个应用类型在一个共同体系统结构上做泛化，使之可以适应不同的客户
COTS 产品复用	系统通过集成已有的应用系统而实现
ERP 系统	为一个机构所开发的一个大型系统，封装了一般业务功能和规则
可配置垂直应用	通用系统的设计可以经过再配置以达到特殊系统客户的需要
程序库	实现通常使用的抽象的类和函数库对复用是有用的
模型驱动工程	软件是表示为领域模型和独立于实现的模型，代码产生于这些模型。详细描述在第5章
程序生成器	生成器系统嵌入一种应用类型的知识，从用户提供的模型来生成该领域中的系统
面向方面的软件开发	在编译程序的时候将共享组件编织到应用的不同位置。详细描述在第21章

图 16-4 支持软件复用的方法



基于生成器的复用

生成器的复用包括把复用概念和知识纳入自动化工具，以及向工具的使用者提供一个简单的方法去整合特有的代码和这个一般的知识。这个方法在特定领域应用中通常是最有效的。在那个领域，已知的解决问题的方法是嵌入在生成器系统中，然后用户选择它来创建一个新的系统。

<http://www.SoftwareEngineering-9.com/Web/Reuse/Generator.html>

关于复用技术的可用情况，在绝大多数情况下，就是指一些软件复用的可能性。是否复用在很大程度上是管理问题而不是技术问题。管理者可能不愿在需求和软件复用之间妥协。他们可能不了解复用相关的风险，而了解本来的开发方式的风险。新的软件开发的开发风险可能很高，管理者也情愿去选择了解的方式而不愿去承担不了解的风险。

16.2 应用框架

早期热衷于面向对象开发的人认为使用面向对象方法的主要好处之一就是在不同的系统中对象可以被重复使用。然而，如前面一节中所说的那样，对象通常粒度太小，而且是面向一个特别应用的。比起重新实现这个对象，它需要更长的时间来了解和适应它。现在已经很清楚的一点是，复用在使用称之为框架的大粒度抽象的面向对象开发过程中得到了最好的支持。

如名字所示，框架是一个一般化的结构，通过对它的扩展来创建更专门的子系统和应用。Schmidt 等（2004）如下定义框架：

“……一个软件人造物的完整集合（人造物如类、对象和组件），它们共同提供一个面向一系列相关应用的可复用的体系结构。”

框架对一般功能提供支持，在所有相似类型的应用中，一般功能很可能被使用。例如，一个用户界面框架可提供事件处理的接口支持，以及包括一个可以用来构造显示的小部件集。然后开发人员专门为特定的应用加入这些特定功能。例如，在一个用户界面框架中，开发人员定义了显示布局，而这个布局是适于正在执行中的应用程序的。

框架支持设计复用，因为它们为应用程序和具体类在系统中的复用提供骨架结构。这个结构是由对象类和它们之间的交互来定义的。类可以直接被复用，也可以利用诸如继承特性来扩展类。

框架实现为用一个面向对象编程语言写的一个具体的和抽象的对象类的集合。因此，框架是具体的语言。在所有经常使用的面向对象程序语言中，都有可用的框架（例如，Java、C#、C++ 和诸如 Ruby 和 Python 动态语言）。一个框架可以包含几个其他的框架，这里的每一个框架都支持应用程序的一部分开发。你可以使用一个框架创建一个完整的应用程序或实现应用程序的一部分，例如图形用户界面。

Fayad 和 Schmidt (1997) 讨论了 3 种类型的框架：

1. **系统基础设施框架** 这些框架支持系统基础设施的开发，这些基础设施包括通信、用户界面和编译器（Schmidt, 1997）。

2. **中间件集成框架** 这些框架是由一组支持组件通信和信息交换的标准和相关对象类构成的。这种类型框架的例子包括 Microsoft 的 .NET 和 Java 中的商业应用组件技术（EJB）。这些框架提供对标准化的组件模型的支持，如在第 17 章所讨论的。

3. **企业应用框架** 这些框架所关心的是专门的应用领域，如通信或金融系统（Baumer 等, 1997）。这些框架嵌入了应用领域知识并支持对最终用户应用的开发。

Web 应用框架（WAFs）是一个较新的和非常重要的框架类型。WAFs 支持动态网站的创建，所以现在已广泛使用。一个 WAFs 的架构通常是以模型 - 视图 - 控制器（MVC）复合模式为基础的，如图 16-5 所示。

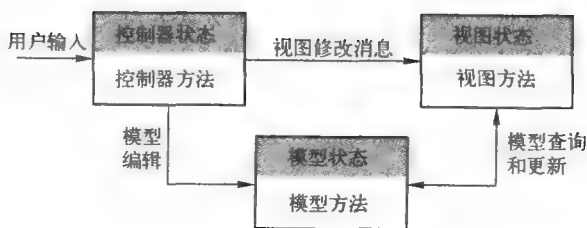


图 16-5 模型 - 视图 - 控制器模式

这个 MVC 模式最初是在 20 世纪 80 年代提出来的。作为一种 GUI 设计方法，它一是允许对象的多个表示形式的存在，二是允许对这些不同表示形态的对象采用不同风格的交互形式。它允许从用户接口到应用的应用程序状态的分离。一个 MVC 框架支持数据表示的不同方式以及允许和每个表示进行交互。当数据在其中一个表示法中被修改时，系统模式也被修改，与每个视图相关的控制器更新它们的表示法。

框架通常是设计模式的实现，正如第 7 章所讨论的那样。举例来说，一个 MVC 框架包括观察者模式、策略模式、组合模式以及其他多个模式。这些模式的介绍可以参考 Gamma 等的著作 (1995)。模式的一般性质，以及它们对抽象和具体类的使用允许它们具有扩展性。如果没有模式，几乎可以肯定框架是不切实际的。

Web 应用框架通常包括一个或多个支持特定应用功能的专业框架。虽然每个框架包括稍有不同的功能，大多数 Web 应用程序框架支持以下功能：

1. **安全性** Web 应用框架可能包括这样的类，以帮助实现用户认证和访问控制，以确保用户只能访问允许在系统中的功能。
2. **动态页面** 提供的这些类来帮助你定义网页模板并且利用从系统数据库中得到的特定数据动态地填充页面。
3. **数据库支持** 框架通常不包括数据库，而是假设一个单独的数据库（例如 MySQL）在使用。框架可以提供给不同的数据库提供虚拟接口的类。
4. **会话管理** 创建和管理会话（大量的用户和系统的交互）的类通常是 WAF 的一部分。
5. **用户交互** 现在大多数 Web 框架提供 AJAX 支持（Holdener, 2008），允许更多的交互网页的创建。

扩展一个框架，不需要改变框架代码，而是要增加具体的类来继承框架中抽象类中的操作。除此之外，回调函数可能需要定义。这些方法的调用是对框架识别事件的响应。Schmidt 等人（2004）称之为“控制反转”。这些框架对象，而不是应用专门的对象，在系统中负责控制。为了响应用户接口、数据库等事件，这些框架对象调用“钩子方法”，然后将之连接到用户提供的功能。具体应用程序的功能则以一种恰当的方式响应这个事件（如图 16-6 所示）。比如，一个框架中有一个方法来控制环境中的鼠标点击事件。这个方法调用钩子方法，你必须配置这个方法来调用相应的应用程序的方法来处理鼠标点击。

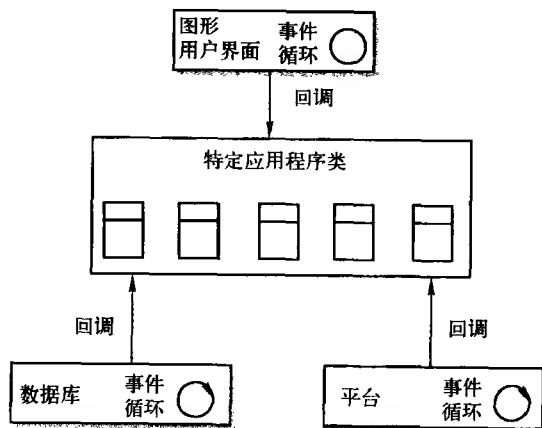


图 16-6 框架中的控制反转

使用框架来构造的应用是采用应用系列概念进行进一步复用的基础，因为这些应用是使用框架构造的，修改系列成员以产生新的系统实例经常是一个简单的过程。它涉及重写添加到框架里的具体的类和方法。

然而，框架通常要比软件产品线更具有一般性，软件产品线着重于应用系统的具体类别。比如，你可以使用基于 Web 的框架来构建基于 Web 的应用程序的不同类型。其中，可能是一个软件产品线，支持基于 Web 的服务台。这个“服务台生产线”以后可能专门来提供服务台的特殊类型的帮助。

框架是复用的一个有效途径，但是引入软件开发过程是非常昂贵的。它们本身就很复杂，学会去使用它们需要花费几个月的时间。评估可用的框架来选择最合适的一个框架是非常困难和昂贵的。调试基于框架的应用程序是困难的，因为你可能不明白那个框架方法是如何相互作用的。这是一个可重复使用的软件的一般问题。调试工具可能会提供有关复用系统组件的信息，而

这是一名开发人员所不知道的。

16.3 软件产品线

复用的最有效的方法之一是创建软件产品线或应用族。产品线是一组应用，它们有相同的体系结构和共享组件，每个具体的应用程序反映不同的要求。核心系统的目的就是进行配置和调整，以适应不同的系统客户的需求。这个可能包括：一些组件的配置，写一些额外的组件，以及调整某些组件以反映新的需求。

通过调整通用应用版本来开发应用程序会有相当大的应用代码得到复用。此外，应用经验通常是可以从一个系统转移到另一个系统的，所以，当软件工程人员加入到开发团队时，他们的学习过程就会缩短。测试得到了简化，因为对于应用程序的大部分测试也是可以复用的，总的开发时间也得到了缩短。

软件产品线总是产生于现有的应用。也就是，机构开发一个应用，然后当一个类似的系统又需要构建时，非严格地在新程序里重复使用已有程序中的代码。当开发其他类似的程序时，要使用同样的过程。然而，由于变更总是要损伤应用结构的，所以随着开发越来越多的新的实例，创造一个新的版本变得越来越困难。所以，应该将设计通用产品线的决定提到议事日程上来。这涉及确定产品实例的一般功能，并包括在基本应用程序中，在以后的开发中会使用。这个基本应用程序是专门构建来简化复用和重新配置的。

应用框架和软件产品线，显然有许多共同点。它们都支持一般的体系结构和组件，以及需要新的开发去创建一个系统的具体版本。这些方法的主要不同如下：

1. 应用框架依靠面向对象的特点，例如继承和多态，来对框架进行扩展。一般地，框架代码是不修改的，可能修改的部分仅限于框架所允许盛纳的组件。软件产品线不必使用面向对象的方法来创建。应用组件是可以修改、删除或重写的。至少在原则上，对可进行的更改没有限制。

2. 应用框架主要侧重于技术支持，而不是特定领域的支持。例如，存在一个可以创建基于 Web 应用程序的应用框架。一个软件产品线通常嵌入了详细的领域和平台信息。例如，可能存在于一个软件产品线，是基于 Web 的用于健康档案管理的应用程序。

3. 软件产品线往往是控制设备的应用程序。例如，有可能存在一系列打印机的软件产品线。这意味着这个软件产品线必须为硬件接口提供支持。应用框架通常是面向软件的，它们几乎不为硬件接口提供支持。

4. 软件产品线是由一系列的相关的应用程序所构成，被同一个机构所拥有。当你创建一个新的应用程序时，你的出发点往往是应用族中最近的成员，而不是通用的核心应用程序。

如果你用一个面向对象的程序语言来开发一个软件产品线，你可能要使用一个应用程序框架作为系统的基础。你可以通过使用它的内在机制扩大具有特定领域组件的框架来创建产品线的核心。

对待开发的软件产品线的各类特化处理需要做的工作包括：

1. **平台特化** 为不同的平台开发应用程序的不同版本。例如，一个应用程序可能存在 Windows、Mac OS、Linux 等各种平台上的版本。在这种情形下，应用程序的功能一般是不会有变化的，只是与硬件和操作系统的接口需要修改。

2. **环境特化** 创建应用的版本来处理特殊的操作环境和外部设备。例如，应急服务系统会存在多个版本，每个版本依赖于车辆通信系统。在这种情况下，系统组件要改变来反映所使用的通信设备的功能。

3. **功能特化** 为不同需求的客户创建不同的应用程序版本。举例来说，图书馆自动化系统

需要根据是用于公众图书馆、查询图书馆还是大学图书馆来修改。在这种情况下，实现功能的组件可能需要修改，新的组件需要添加。

4. **过程特化** 调整系统使之与特殊的业务过程配套。例如，订单系统需要调整来应对公司内的集中式订购过程以及其他公司的分布式过程。

一个软件产品线体系结构往往反映了一个普遍的、特定的应用程序结构风格或模式。例如，考虑设计用于处理车辆调度和应急服务的产品线系统。此系统操作员接听事故电话，寻找适当车辆并派发到事故现场。此类系统的开发者可以将其调整成适应警务、消防和急救服务。

此车辆调度系统是资源管理系统的一个例子，这种资源管理系统的应用体系结构如图 16-7 所示。从图 16-8 可以看到如何对四层结构进行实例化，图中给出了应该出现在车辆调度系统产品线中的一些模块。产品线系统中每一层中的组件是：

1. 在交互层，有提供给操作者显示的界面和与通信系统之间的接口。
2. 在 I/O 管理层（层 2），有处理操作者权限的组件，生成事件和车辆配发报告的组件，支持地图输出和路径规划的组件，还有提供给操作者查询系统数据库的机制的组件。

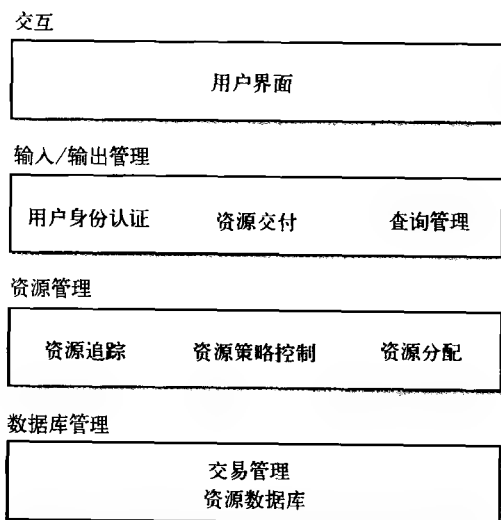


图 16-7 资源分配系统的体系结构

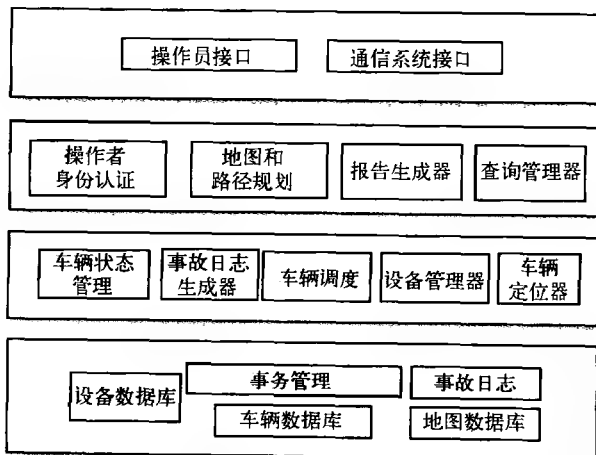


图 16-8 车辆调度系统的产品线体系结构

3. 在资源管理层（层3），有定位和调度车辆的组件，更新车辆和设备状态的组件，以及记录事故细节的组件。

4. 在数据库层，除了有一般的事务管理支持组件外，还有单独的车辆、设备和地图数据库。

为了创建一个此系统的特殊版本，需要修改单个组件。例如，警察部门有大量车辆，但是车辆的种类很少，而消防队有众多类型的专门车辆，所以当为不同的服务实现系统时，你需要定义不同的车辆数据库结构。

图 16-9 给出通过扩展应用产品线来创建一个新应用的步骤。包含在这样一个一般过程中的步骤是：

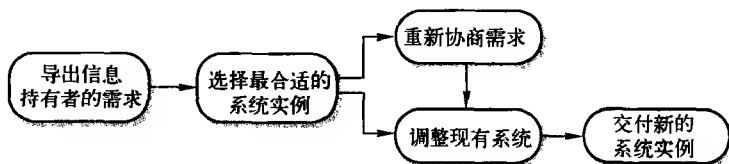


图 16-9 产品实例开发

1. **导出信息持有者需求** 这可以基于一般的需求工程。不过，因为系统已经存在，它一般包括演示和试验系统，把需求表示为对所需功能的修改。

2. **从现存系统中选择最符合要求的需求** 对需求进行分析，进而选择最合适的产品成员作为修改目标。

3. **重新协商需求** 因为需要对现有系统出现的更多细节进行更改，而项目已经规划好了，因而可能有必要对需求重新协商一下，将需要的变更减到最少。

4. **调整现有系统** 为系统再开发一些新模块的同时，改写现有的系统模块来适应新的需求。

5. **交付新族成员** 将应用系列（族）的新成员交付给客户。在这一阶段，应该记下它的主要特征，以便它可以在未来的系统开发中使用。

当创建产品线的新成员时，可能必须在复用和满足需求之间做出妥协，一是要尽量多地复用一般应用的内容，另一个是要尽量满足信息持有者的详细需求。系统需求越详细，现有组件满足需求的希望就越小。不过，如果信息持有者对需求要求是有弹性的，愿意限制系统的修改幅度，则系统就能更快速和低成本地交付了。

软件产品线的设计是用来配置的。此配置可以包括添加组件到系统中或是从系统中删除组件，为系统组件定义参数和约束，包含业务过程知识等。这个配置可以发生在开发过程的不同阶段：

1. **设计时配置** 开发软件的机构通过开发、选择或调整为顾客创建新系统的组件来修改通用产品线。

2. **部署时配置** 先设计一个通用系统，然后由客户或者是专家与客户一起对系统进行配置。此系统采用一组配置文件来保存配置信息，客户的特殊需求知识和系统运行环境知识都嵌入在这样的一组配置文件中。

当一个系统在设计阶段进行配置时，供应商从通用系统或一个存在的产品实例开始。通过修改和扩展系统中的模块，创建一个可以交付给客户的能满足客户所要求功能的专门系统。此方法总是包括改变和扩展核心系统的源代码，这样就可以获得比部署时配置更大的适应性。

部署时配置包括使用一个配置工具来创建一个特定的系统配置，它被记录在配置数据库中或一个配置文件中（见图 16-10）。当执行时，这个正在执行的系统可能会查询这个数据库，所以其功能会根据其执行上下文而特殊处理。

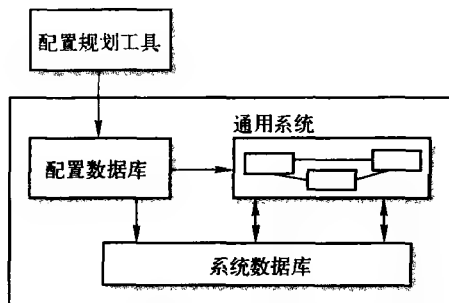


图 16-10 部署时配置

在一个系统中，可能会提供几个层次的部署时配置：

1. 组件选择，即在系统中选择提供所需功能的那些模块。例如，在一个病人信息系统中，你可以选择一个图像管理组件，允许你将医疗图像（X 射线，CT 扫描等）和病人的医疗记录联系起来。

2. 工作流和规则定义，即你定义工作流（信息是如何一步一步加以处理的）和有关应用于由用户输入的或由系统生成的信息的验证规则。

3. 参数定义，即对特定的系统参数定义取值，这些参数反映了你正在创建的应用程序的实例。例如，你可以给定数据输入字段的最大长度，这可以由用户输入或由系统硬件的特性所产生。

部署时配置可以非常复杂，可能需要许多个月来配置一个客户系统。大型配置的系统可以支持通过提供软件工具的配置过程，诸如提供配置规划工具，来支持配置过程。16.4.1 节将进一步讨论部署时配置。这涵盖了 COTS 的系统的复用，必须对它进行配置，以工作在不同的运行环境。

使用系统中的现有的部署时配置设施去开发一个新系统版本是不可能的。这时就需要设计时配置了。然而，随着时间的推移，当你创造了几个具有同等功能的族成员时，你可以决定重构核心产品线，来包括已在多种应用族成员中实现的功能。然后，当系统部署时，你可以使这一新的功能是可配置的。

16.4 COTS 产品的复用

商业现货（COTS）产品是无需修改系统的源代码，就能适应不同客户的需求即可使用的软件系统。事实上，所有的桌面软件和很大一部分服务器产品都是 COTS 软件。由于这些软件是为通用所设计的，它总是包括很多特性和功能，因此，它有可能被重复使用在不同环境中和不同的应用程序的一部分中。Torchiano 和 Morisio（2004）还发现，使用的开源产品往往是 COTS 产品。也就是说，使用开源系统，是不需要修改和查看源代码的。

COTS 产品靠使用内置的配置机制来适应需求，内置的配置机制允许调整系统的功能以适应特定客户的需求。例如，在一个医院病人记录系统中，为不同类型的病人，可能定义独立的输入形式和输出报告。其他配置功能可能允许系统接受插件，它用来扩展功能或检查用户输入，以确保它们是有用的。

在过去 15 年左右的时间里，这个软件复用的方法被很多大公司广泛采用，因为它提供了比传统软件开发更大的好处：

1. 随着其他复用类型的使用，一个可靠的系统的更快速部署是可能的。
2. 看到应用程序提供哪些功能是可能的，因此很容易判断它们是否合适。其他公司可能已

经在使用这样的应用程序，所以系统经验是可用的。

- 3. 使用已经存在的软件可避免一些开发风险。然而，这个方法存在着风险，稍后将讨论之。
- 4. 业务部门可以只专注于他们的核心活动，而不必投入大量的资源到 IT 系统的开发。
- 5. 随着运行平台的进化，技术更新可能会简化，因为这些都是现成的产品供应商应负的责任，而不是客户。

当然，这个软件工程方法存在它自己的问题：

- 1. 需求通常都必须调整，以反映 COTS 产品的操作功能和模式。
- 2. COTS 产品可能基于这样的假设：在实际中它们是不可能改变的。因此，客户必须调整他们的业务，以反映这些假设。
- 3. 为企业选择合适的 COTS 系统可能是一个艰难的过程，特别是许多 COTS 产品都没有详细的描述。作出错误的选择可能是灾难性的，因为它可能无法使新的系统按照所需要的那样工作。
- 4. 有可能是缺乏本地的专家以支持系统的开发。因此，客户必须依赖于供应商和外部顾问以得到有关开发的建议。这个建议可能是有偏见的和面向销售的产品和服务的，而不是满足客户的真正需要。
- 5. 该 COTS 产品供应商控制着系统的支持和进化。他们可能会倒闭，被接管，或可能做出改变，为客户带来困难。

基于 COTS 的软件复用，已经越来越普遍。新的业务信息处理系统，绝大部分是使用 COTS 建成的，而不是使用面向对象的方法。尽管用这种方法进行系统开发可能存在某些问题（Tracz, 2001），成功案例（Baker, 2002；Balk 和 Kedia, 2000；Pfarr 和 Reis, 2002）说明基于 COTS 的复用减少了部署这一系统的人力和时间。

存在两种类型的 COTS 产品复用，即 COTS 解决方案系统和 COTS 集成系统。COTS 的解决方案系统包括一个通用的来自一个单一的供应商应用程序，根据客户的需求来配置应用程序。COTS 的集成系统涉及集成两个或两个以上的 COTS 系统（可能来自不同厂商）来创建一个应用系统。图 16-11 总结了这些不同的方法之间的差异。

COTS 解决方案系统	COTS 集成系统
独立的产品，提供一个客户要求的功能	集成多个异构系统产品来提供个性化的功能
基于围绕一个通用的解决方案和标准化的过程	可为客户过程开发灵活的解决方案
开发侧重于系统配置	开发侧重于系统集成
系统供应商负责维持	系统拥有者负责维持
系统供应商提供系统平台	系统拥有者提供系统平台

图 16-11 COTS 解决方案系统和 COTS 集成系统

16.4.1 COTS 解决方案系统

COTS 的解决方案是通用的应用系统，可能是设计来支持特定的业务类型和业务活动，有时可能是支持一个完整的商业企业。例如，为牙医开发的一个 COTS 系统，用来处理预约、牙齿记录、病人召回等。更大规模来说，企业资源规划（ERP）系统可支持在一家大公司的所有的生产、订货和客户关系管理活动。

特定领域的 COTS 解决方案系统，比如支持一个商业功能（例如文件管理）的系统，其提供的功能，很可能是由一系列的潜在用户所需要的。然而，它们也包含内嵌的关于用户如何工作的假设，这可能会在特定情况下引起问题。例如，一个支持在大学里学生注册的系统可能假设学生

在一个大学只能注册一个学位。然而，如果大学合作来提供联合学位，那么要在这个系统中来表现出来几乎是不可能的。

企业资源规划（ERP）系统如那些由 SAP 和 BEA 所生产的系统，是大型集成系统，设计来支持如订单和发票、库存管理以及制造调度这些业务过程的（O'Leary, 2000）。它们的配置过程包含收集有关客户的业务和业务过程的详细信息，然后将此信息嵌入到配置数据库中。这通常需要配置符号系统和工具的详细知识，并需要专家与系统客户一道工作完成配置任务。

一个通用 ERP 系统包含大量模块，这些模块可以以不同的方式来组合为一个客户创建一个系统。配置过程包括选择要包括的模块，配置这些单个模块，定义业务过程和业务规则，定义系统数据库的结构和组织。支持各种业务功能的一个 ERP 系统总体结构模型如图 16-12 所示。

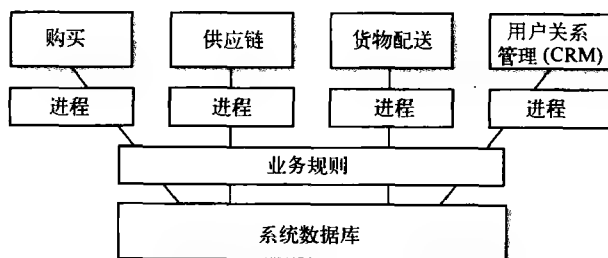


图 16-12 ERP 系统的体系结构

这个结构的关键特征是：

1. 支持不同的业务功能的众多模块。这些都是大粒度模块，支持整个部门或业务部门。在图 16-12 所示的例子，已经被选为包括在该系统的模块是一个支持供应链管理的模块，一个支持货物交付的物流模块和一个维护客户信息的客户关系管理（CRM）模块。

2. 定义好的一组业务流程，与每个模块相关联，涉及在那个模块中的若干活动。例如，可能有一个订购流程的定义，它定义了如何创建和核准订单。这将指定参与订货的角色和活动。

3. 一个共同的数据库拥有所有相关业务功能的信息。这意味着不必在不同业务部分中复制诸如用户详细信息。

4. 一个适用于数据库中的所有数据的业务规则。因此，当数据从一个函数输入时，这些规则应确保它与其他函数所需要的这个数据相一致。例如，可能有一个业务规则，所有费用报销必须由比需要报销的人更高一级的人员批准。

大多数大公司都使用 ERP 系统来支持它们的某些或全部功能。ERP 系统因此是使用最广的软件复用的形式。然而，这种复用方法的明显的限制是系统的功能受限于通用核心的功能。而且，公司的运行必须表述为系统的配置语言，而业务概念和配置语言所支持的概念之间是会在某些不一致的地方的。

例如，在出售给大学的 ERP 系统中，客户的概念就必须定义。在配置这个系统时，遇到了很大的困难。大学有多种类型的客户，例如学生、科研基金代理机构、教育慈善机构等，他们中的每一个都有不同的特点，都完全不同于商业客户（例如，购买产品或服务的一个人或一个单位）。在系统和客户所使用的业务模型上的严重不匹配使得 ERP 系统不能满足客户真实需要的可能性非常大（Scott, 1999）。

这两个特定领域的 COTS 产品和 ERP 系统通常需要大量的配置，以适应每一个机构的安装要求。这个配置可能涉及：

1. 从系统中选择所需要的功能（例如，决定包括什么样的模型）。
2. 建立数据模型，定义在系统的数据库中机构的数据是如何组织的。

3. 定义适用于数据的业务规则。
4. 定义和外部系统的预期的交互。
5. 设计由系统生成的输入形式和输出报告。
6. 设计新的业务流程, 符合系统支持的基本的过程模型。
7. 设置参数, 它定义了是如何在基础平台上进行部署的。

一旦配置设置完成, 一个 COTS 解决方案系统就可以进行测试了。当配置系统时, 而不是使用传统的编程语言来开发时, 测试是一个重大问题。因为这些系统是使用一个可信赖的平台来开发的, 明显的系统错误和崩溃是相当少的。相反的, 这些问题往往是很微妙的, 它涉及业务之间的过程和系统配置的相互作用。这些可能只能被最终用户检测出来, 因此可能不会在系统测试过程中发现。此外, 不能使用依靠测试框架如 JUnit 框架的自动化单元测试。这个基本系统是不可能支持任何形式的测试自动化的, 可能没有完整的系统规范可用于来派生出系统测试。

16.4.2 COTS 集成系统

COTS 集成系统是包括两个或更多 COTS 产品的应用程序, 或者有时候是遗留应用系统。当没有单个的 COTS 产品满足你的需求时, 或是当你想融入一个新的 COTS 的系统到已经在使用的产品中时, 可以使用这个方法。如果这些接口已经定义好了的话, 这个 COTS 产品可以通过应用程序编程接口或服务的接口来交互。此外, 它们可能是通过连接一个系统输出到另一个系统的输入或更新的现成应用程序使用的数据库而组成的。

通过使用 COTS 产品进行系统开发, 你需要做出多个设计选择:

1. **哪个 COTS 产品提供最为恰当的功能?** 通常, 会有几个可用的 COTS 产品, 它们需要以不同的方式整合进来。如果你还没有使用 COTS 产品的经验, 决定哪个产品是最为合适的是一件很困难的事。
2. **数据如何交换?** 一般来讲, 不同产品通常使用独特的数据结构和格式, 你需要编写适配器来完成从一个表示到另一个表示的转换。这些适配器是运行时系统, 与 COTS 产品一起工作。
3. **实际使用产品的哪些特性?** 绝大多数 COTS 产品有比你所需要的多得多的功能, 而且产品间功能上的重复性很大。你需要决定哪个产品的哪个特性最为适合你的需要。如果可能的话, 你还应该不允许对未使用的功能访问, 因为这可能干扰正常的系统操作。阿丽亚娜 5 号火箭 (Nuseibeh, 1997) 的第一次飞行失败, 是由于在惯性导航系统中的一个失败所引起的, 这个惯性导航系统是复用阿丽亚娜 4 号系统的。然而, 那个失败的功能在阿丽亚娜 5 号中是没有需求的。

作为对 COTS 集成的说明, 考虑下面的情形。有一个大型机构计划开发一个采购系统, 允许工作人员从他们的个人计算机上下订单。通过在机构中引入此系统, 公司估计每年可以节省 500 万美元。通过集中采购, 新的采购系统能确保总是向提供的价格最为便宜的供货商下订单, 而且应该减少订单所相关的一系列文书工作的费用。与手工系统相同, 这包括从供货商处选择可用商品, 创建一个订单, 在订单得到批准后, 将订单发送到供货商, 接收货物并确定应该支付的内容。

公司已存在一个订货系统, 由采购办公室使用。此订单处理软件已经集成了一个已存在的货品计价和发送系统。为了创建新的订货系统, 他们将遗留下的系统与一个基于 Web 的商务平台以及一个处理用户间通信的电子邮件系统集成在一起。最后的使用 COTS 的采购系统的结构如图 16-13 所示。

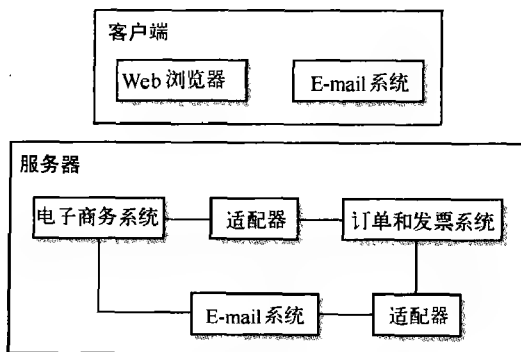


图 16-13 COTS 集成的获得系统

该采购系统是基于客户端/服务器模式的，在客户端，使用的是标准的 Web 浏览和 E-mail 软件。在服务器端，电子商务平台通过适配器与现有的订货系统集成在一起。电子商务系统有其自己的订单格式和交货形态等，这些都需要转变为订单系统所使用的格式。电子商务系统与 E-mail 系统是固定集成在一起的，能发送通知单给客户，但是订单系统却没有这个设计。为此，需要写另一个适配器把订货系统中的通知单转换成 E-mail 消息。

使用 COTS 集成的方法有时能节省数月甚至数年的工作量，开发和部署一个系统的时间可以大大降低。在上面所描述的采购系统在非常大的公司中的实现和部署花了 3 个月的时间而不是像他们估计的需要在 Java 中开发 3 年。

如果使用面向服务的方法，COTS 集成可以被简化。本质上讲，面向服务的方法意味着允许你通过标准的服务接口访问应用系统中的功能。有些应用程序提供服务接口，但是有时候，这个服务接口必须由系统集成者自己实现。从本质上讲，你要一个包装程序，来隐藏应用程序并提供外部可见的服务（见图 16-14）。这种方法对那些必须集成较新的应用系统的遗留系统是特别有价值的。

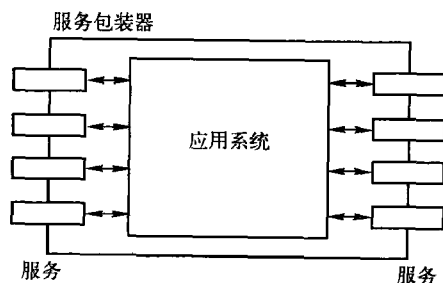


图 16-14 应用程序包装

原则上讲，集成一个大规模的商业现货系统与集成其他组件没有什么两样。必须了解系统的接口并且只使用它们与该软件通信；必须在满足专门需求和快速开发及复用之间做出权衡；必须设计一个系统体系结构来盛纳这些商业现货系统，使之一起工作。

然而，这些产品通常都是一些大型系统并且经常作为一个独立系统出售，由此产生了一些额外问题。Boehm 和 Abts (1999) 讨论了 4 个重要的 COTS 系统集成问题：

1. 对功能和性能缺乏控制 虽然发布的产品接口似乎提供了所需要的设施，但这些可能没有正确地实现或者是性能较差。产品中可能存在一些隐藏的操作干扰它的正常功能。修补这些问题可能成为集成 COTS 产品时的一个首要问题，但这只是集成者所要关心的事，提供商是不会真正关心的。用户要想复用 COTS 产品，就要查清问题的影响范围。

2. **商业现货系统间的互操作问题** 有时让各种 COTS 产品协同工作是件困难的事, 因为每种产品都有其对使用的假定条件。Garlan 等 (1995) 报告了他们在集成 4 种商业现货产品的经验, 发现其中的 3 个产品是基于事件的, 但每一个都有其不同的事件模型。每个系统假定对事件序列的访问入口都是唯一的。结果是, 集成非常困难。项目需要的工作量是原先预计的 5 倍, 原先预计 6 个月完成该项目, 最后花了两年才完成。10 年后在对他们工作的回顾性分析中, Garlan 等 (2009) 得出的结论是, 他们所发现的集成问题仍然没有得到解决。Torchiano 和 Morisio 发现, 在一些 COTS 产品中缺乏对标准的遵守, 这意味着集成比预期的更加困难。

3. **无法控制系统的进化** 商业现货产品的厂商根据他们自身感受的市场压力来决定对系统的变更。尤其对 PC 产品, 新版本的发布非常频繁, 而且这些新版本并不完全与旧的版本兼容。新版本可能有附加的不必要的功能, 而早先的版本可能难以得到和得不到支持。

4. **COTS 产品厂商的支持** 商业现货产品厂商能提供的支持程度很不一样。开发者没有可能看到其源代码以及详细的系统文档, 因而当出现问题时, 来自厂商的支持就变得尤为重要。虽然厂商可能承诺提供支持, 但是变化的市场和经济环境可能使他们难以兑现这些承诺。举例来说, 一个 COTS 系统厂商可能决定停止某个产品的生产, 因为该产品的需求已经很小, 或者是公司已经由别的公司接管, 新公司不想对所有已被收购产品都提供支持。

Boehm 和 Abts 认为在许多情况下, 使用 COTS 产品比不使用这些产品在系统维护和进化方面的成本要有所提高。所有这些困难都是生命周期问题, 它们不仅仅影响初始的系统开发。参与系统维护的人员与最初的系统开发人员之间的差距越大, 在 COTS 产品集成时真正出现困难的可能性就越大。

要点

- 现在大多数新的业务软件系统是通过复用以往已实现的系统的知识和代码来开发的。
- 复用软件有很多种方法。这些包括从对库中类和方法的复用到对完整应用系统的复用。
- 软件复用的好处是较低的成本、较快速的软件开发以及较小的风险。同时, 系统可靠性得到增加, 而且能让专家腾出精力将其专门技术更有效地用于可复用组件的设计上。
- 应用框架是这样的一些具体和抽象的对象的集合, 这些对象的设计是要通过特化和添加新对象使之得以复用。他们通常通过设计模式来体现良好的设计实践。
- 软件产品线是一组相关应用, 从一个或多个基础应用衍生而来。通用系统可以调整和特化来满足在功能上的特殊需要, 或者是适应不同的平台或运行配置。
- COTS 产品复用是关于对大型现货产品系统的复用。这些产品提供了大量的功能, 对它们的复用能极大地降低成本和节省开发时间。开发一个系统, 可以通过配置一个单独的、通用的 COTS 产品或是通过集成两个或更多的 COTS 产品。
- 企业资源规划 (ERP) 系统是大型 COTS 复用的例子。特殊 ERP 系统是通过将通用系统配置创建的, 这种配置可以在部署时利用客户的业务进程和规则信息进行。
- 基于 COTS 产品的复用的潜在问题包括对功能和性能缺乏控制, 同时缺乏对系统进化的控制, 需要来自外部供应方的支持, 以及在确保系统能互操作上的困难。

进一步阅读材料

《Reuse-based Software Engineering》, 这是一本全面讨论软件复用不同方法的书, 涵盖了技术复用问题和管理复用过程 (H. Mili, A. Mili, S. Yacoub and E. Addg, John Wiley & Sons, 2002)。

《Overlooked Aspects of COTS-Based Development》, 这是一篇非常有趣的文章, 它讨论了采用基于 COTS 的方法的开发人员的一份调查和他们所遇到的问题 (M. Torchiano and M. Morisio, IEEE

Software, 21(2), March – April 2004)。http://dx.doi.org/10.1109/MS.2004.1270770。

《Construction by Configuration: A new Challenge for Software Engineering》，这是一篇邀稿，作者在其中讨论了通过配置现有的系统来构建一个新的应用程序的问题和困难(I. Sommerville, Proc, 19th Australian Software Engineering Conference, 2008)。http://dx.doi.org/10.1109/ASWEC.2008.75。

《Architectural Mismatch: Why Reuse Is Still So Hard》，这篇文章回顾了早前的一篇关于复用和集成多个 COTS 产品系统的问题的文章。作者得出结论，虽然取得了一些进步，仍然还有与单个系统的设计者作出的假设相冲突的问题(D. Garlan et al., IEEE Software, 26(4), July – August 2009)。http://dx.doi.org/10.1109/MS.2009.86。

练习

- 16.1 阻碍软件复用的技术和非技术因素有哪些？从你自己的经验来看，你复用过很多软件吗？如果没有，那又是为什么？
- 16.2 解释一下为什么通过复用已有的软件所节省的成本并不是简单地与所使用的组件规模成比例。
- 16.3 给出你认为不应该使用软件复用的 4 种情况。
- 16.4 解释“控制反转”是什么意思。解释如果集成两个独立的系统，其中这两个系统是使用同样的应用程序框架来开发的，为什么这个方法会产生问题。
- 16.5 使用第 1 章和第 7 章的气象站系统，为这个应用族提出一个产品线体系结构，该应用族处理远程监控和数据收集。你应该展现一个分层模型的架构，显示在每个层可能包含的组件。
- 16.6 大多数桌面软件，如文字处理软件，可以以许多不同的方式配置。检查你经常使用的软件，列出为那个软件进行配置的选项。说明用户在配置这个软件时遇到的困难。如果你使用 Microsoft office 或是 Open office，这些是做这个练习的很好的例子。
- 16.7 为什么许多大公司选择 ERP 系统作为他们机构的信息系统的基本系统？在一个机构中部署一个大规模的 ERP 系统，可能出现什么问题？
- 16.8 找出在通过使用 COTS 进行系统构造是会发生 6 种风险。公司能采取哪些步骤来降低这些风险。
- 16.9 解释为什么在通过集成 COTS 产品构造系统时适配器总是需要的。说出在写连接两个现成应用产品的适配器软件时可能出现的 3 个实际问题。
- 16.10 软件复用涉及很多版权和知识产权问题。如果一个客户出资让软件开发商开发某系统，谁将拥有所开发代码的复用权？软件开发商有权使用这些代码作为一种一般组件的基础吗？应该使用什么支付机制来补偿可复用组件的提供者？讨论这些问题和其他的有关软件复用的道德争议。

参考书目

- Baker, T. (2002). 'Lessons Learned Integrating COTS into Systems'. *Proc. ICCBSS 2002 (1st Int. Conf on COTS-based Software Systems)*, Orlando, Fla.: Springer, 21–30.
- Balk, L. D. and Kedia, A. (2000). 'PPT: A COTS Integration Case Study'. *Proc. Int. Conf. on Software Eng.*, Limerick, Ireland: ACM Press, 42–9.
- Baumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D. and Zullighoven, H. (1997). 'Framework Development for Large Systems'. *Comm. ACM*, 40 (10), 52–9.
- Boehm, B. and Abts, C. (1999). 'COTS Integration: Plug and Pray?' *IEEE Computer*, 32 (1), 135–38.
- Brownsword, L. and Morris, E. (2003). 'The Good News about COTS'. <http://www.sei.cmu.edu/news-at-sei/features/2003/1q03/feature-1-1q03.htm>
- Cusamano, M. (1989). 'The Software Factory: A Historical Interpretation'. *IEEE Software*, 6 (2), 23–30.

- Fayad, M. E. and Schmidt, D. C. (1997). 'Object-oriented Application Frameworks'. *Comm. ACM*, **40** (10), 32–38.
- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley.
- Garlan, D., Allen, R. and Ockerbloom, J. (1995). 'Architectural Mismatch: Why Reuse is so Hard'. *IEEE Software*, **12** (6), 17–26.
- Garlan, D., Allen, R. and Ockerbloom, J. (2009). 'Architectural Mismatch: Why Reuse is Still so Hard'. *IEEE Software*, **26** (4), 66–9.
- Griss, M. L. and Wosser, M. (1995). 'Making reuse work at Hewlett-Packard'. *IEEE Software*, **12** (1), 105–7.
- Holdener, A. T. (2008). *Ajax: The Definitive Guide*. Sebastopol, Calif.: O'Reilly and Associates.
- Jacobson, I., Griss, M. and Jonsson, P. (1997). *Software Reuse*. Reading, Mass.: Addison-Wesley.
- Matsumoto, Y. (1984). 'Some Experience in Promoting Reusable Software: Presentation in Higher Abstract Levels'. *IEEE Trans. on Software Engineering*, **SE-10** (5), 502–12.
- McIlroy, M. D. (1968). 'Mass-produced software components'. *Proc. NATO Conf. on Software Eng.*, Garmisch, Germany: Springer-Verlag.
- Nuseibeh, B. (1997). 'Ariane 5: Who Dunnit?' *IEEE Software*, **14** (3), 15–6.
- O'Leary, D. E. (2000). *Enterprise Resource Planning Systems: Systems, Life Cycle, Electronic Commerce and Risk*. Cambridge, UK: Cambridge University Press.
- Pfarr, T. and Reis, J. E. (2002). 'The Integration of COTS/GOTS within NASA's HST Command and Control System'. *Proc. ICCBSS 2002 (1st Int. Conf on COTS-based Software Systems)*, Orlando, Fla.: Springer, 209–21.
- Schmidt, D. C. (1997). 'Applying design patterns and frameworks to develop object-oriented communications software'. In *Handbook of Programming Languages, Vol. 1*. (ed.). New York: Macmillan Computer Publishing.
- Schmidt, D. C., Gokhale, A. and Natarajan, B. (2004). 'Leveraging Application Frameworks'. *ACM Queue*, **2** (5 (July/August)), 66–75.
- Scott, J. E. (1999). 'The FoxMeyer Drug's Bankruptcy: Was it a Failure of ERP'. *Proc. Association for Information Systems 5th Americas Conf. on Information Systems*, Milwaukee, WI.
- Torchiano, M. and Morisio, M. (2004). 'Overlooked Aspects of COTS-Based Development'. *IEEE Software*, **21** (2), 88–93.
- Tracz, W. (2001). 'COTS Myths and Other Lessons Learned in Component-Based Software Development'. In *Component-based Software Engineering*. Heineman, G. T. and Councill, W. T. (ed.). Boston: Addison-Wesley, 99–112.

基于组件的软件工程

目标

本章的目标是阐述一种基于可复用的、标准化组件组合的软件复用方法。读完本章，你将了解以下内容：

- 基于组件的软件工程是有关基于组件模型的标准化组件开发以及将其组成应用系统等内容；
- 了解组件和组件模型的含义；
- 了解面向复用的 CBSE 过程的主要活动和基于复用的 CBSE 过程；
- 了解在组件合成过程中出现的困难和问题。

正如第 16 章中所讨论的，很多新的业务系统是通过配置现成的系统来开发的。然而，当一个公司因为现货系统不能满足他们的需求时，就不能使用现成的系统，他们因而需要进行专门开发。对于定制软件，基于组件的软件工程是一种有效的、面向复用的开发新的企业系统的方法。

基于组件的软件工程出现于 20 世纪 90 年代末期，是软件系统开发的基于复用的方法。它的产生是由于设计者们在使用面向对象的开发过程中所受到的挫折，这种挫折来自于面向对象开发不能够实现像人们最初所期待的那样能完成广泛的复用。单个对象类有太多细节且太特殊，通常需要在编译时间与应用绑定。我们必须拥有对类的详细的知识以便于应用它们。一般来讲这意味着不得不掌握组件的源代码。这意味着，销售和发布对象作为单独的可复用的组件实际上是不可行的。

组件是比对象更高层次的抽象，是由它们的接口来定义的。它们一般比对象大，所有的实现细节对其他组件是隐藏的。CBSE 是定义、实现和集成或组合松散耦合的独立组件成系统的过程。它之所以成为重要的软件开发方法是因为软件系统变得更大、更复杂，且用户要求开发更可靠、发布和部署更快的软件。我们处理这种复杂性，并更快交付软件的方法是复用软件组件而不是重新实现软件组件。

基于组件的软件工程的要素有：

1. 独立组件由它们的接口完全定义。在组件接口与组件实现之间应该被明确地分离开来，这意味着用另一种方法实现组件代替其他的实现时系统不发生改变。

2. 组件标准使组件集成变得更为容易。这些标准包含在组件模型之中，在最低程度上定义了组件接口应该如何定义，如何实现组件间的交互。一些模型定义了应该由所有相容组件所实现的那些接口。如果组件构造符合标准，则它们的操作独立于编程语言。用不同语言编写的组件可集成在同一个系统中使用。

3. 中间件为组件集成提供软件支持。为了使独立的、分布式的组件一起工作，需要有处理组件之间通信的中间件。支持组件的中间件可以有效地处理低层的问题，并允许我们集中精力来处理与应用相关的问题。此外，支持组件的中间件可以提供对资源分配、事务管理、信息安全及并发的支持。

4. 开发过程是一种适应于基于组件的软件工程的开发过程。你需要一个开发过程，它允许

根据可用的组件功能对需求进化。17.2 节将讨论 CBSE 的开发过程。



CBSE 问题

CBSE 现在是软件工程中的主流方法——它是建立一个系统的好的方法。然而，当作为复用的方法使用时，尤其是当它们和其他的组件整合时。问题包括组件诚信，组件验证，需求折中和预测组件的属性。

<http://www.SoftwareEngineering-9.com/Web/CBSE/problems.html>

基于组件的开发体现了良好的软件工程实践。它不仅对于使用组件来设计系统是有意义的，即使你不是复用这些组件，而是开发这些组件，也是很有意义的。基本的 CBSE 是支持构造易理解和可维护软件的可靠和有效的设计原则：

1. 组件是独立的，因此它们不会彼此的操作。组件实现的细节是隐藏的。组件的实现的变化可以不影响系统其他部分。
2. 组件通过良好定义的接口进行交互，如果这些接口能得到保持的话，组件便可以更换为另一个有更多功能或由更先进功能的组件所替代。
3. 组件基础设施提供一系列可用在应用系统中的标准服务。这个减少了要开发的新代码的量。

CBSE 最初的动机是要支持复用和分布式软件工程。组件被看做是一个软件系统的元素，它可以由在不同的计算机上运行的其他组件使用远程过程调用机制来访问。每个系统复用一个组件必须包含其该部分的副本。这种组件的概念扩大了分布式对象的概念，如在分布式系统模型的定义，如 CORBA 的资料参考 (Pope, 1997)。已经开发了几种不同的协议和标准支持这一组件观点，例如 Sun 公司的 EJB，微软的 COM 和 .NET，CORBA 的 CCM (Lau 和 Wang, 2007)。

实际上，这些多重标准阻碍了 CBSE 的采用。对于开发的组件来说，使用不同的方法来共同工作是不可能的。针对不同的平台开发的组件，例如 .NET 或 J2EE，不能互操作。而且，提议的这些标准和协议非常复杂，很难理解。这也是采用组件的一个障碍。

针对这些问题，发展了组件作为一项服务的概念和提出这些标准来支持面向服务的软件工程。作为一项服务的组件和组件的原始概念之间最大的不同之处是，服务是独立的实体，在使用它们的程序的外部存在。当你创建一个面向服务的系统时，你是引用外部服务，而不是在你的系统中包含进一个该服务的拷贝。

面向服务的软件工程在第 19 章中讨论，因此是一种基于组件类型的软件工程。相比原先在 CBSE 中建议的来说，它使用了更简单的组件概念。从开始，它就已经是由标准来驱动的。在这样的情况下，即基于 COTS 的复用是不切实际的时候，面向服务的 CBSE 对于业务系统的开发来说，逐渐变成了主要的方法。

17.1 组件和组件模型

在 CBSE 业界，一般观点认为组件是一个独立的软件单位，可以与其他组件构成新的软件系统。然而，不同的人对软件组件提出了不同的定义。Council 和 Heineman (2001) 将组件定义为：

组件是一种软件元素，与某个组件模型要求相一致，按照组成标准无需修改即可独立进行部署和组合。

这个定义实质上是基于标准的——遵循这些标准的软件单元即为一个组件。然而，Szyperski (Szyperski, 2002) 关于组件的定义却并不提及标准，而是将重点放在了组件的关键特征上：

组件是具有合同定义的接口和显式的上下文依赖，可独立进行部署的并服从于第三方的组成的软件单元。

这两个定义都是基于组件作为一个元素的概念，它是包含在一个系统中，而不是把组件作为一项服务被系统引用。然而，它们还是和服务作为一个组件的概念相兼容的。

Szyperski 同时指出组件没有从外部可观察的状态。这意味着组件的拷贝彼此是很难区分的。然而，一些组件模型，例如企业 Java Beans 模型允许有状态的组件。因此这些与 Szyperski 的组件定义不一致。虽然无状态的组件使用起来更简单，但是存在一些系统，有状态的组件的使用更加方便，并且减少了系统的复杂性。

上面定义的共同性在于它们都认为组件是独立的并且是系统最基本的组成单元，通过组合这些定义我们可以得出组件的一个更好的定义。图 17-1 所示为用于 CBSE 的组件的最重要的特征。

组件特性	描 述
标准化	组件标准化意味着使用在 CBSE 过程中的组件必须符合某种标准化的组件模型。此模型会定义组件接口、组件元数据、文档管理、组成以及部署
独立性	组件应该是独立的，它应该可以在无其他特殊组件的情况下进行组合和部署。如果在某些情况下组件需要外部提供的服务，应该在“需要”接口描述中显式地声明
可组合性	对于可组合的组件，所有外部交互必须通过公开定义的接口进行。另外，它还必须提供对自身信息的外部访问，例如它的方法和属性
可部署性	为使之可部署，组件需要是自包含的，它必须是能作为一个独立实体在提供其组件模型实现的组件平台上运行。因而意味着组件总是二进制形式的且无需在部署前编译。如果一个组件实现为一项服务，它不必由用户来部署，而是由服务的提供者来部署
文档化	组件必须是完全文档化的，这样所有用户能确定是否组件满足他们的需要。应该定义所有组件接口的语法甚至语义

图 17-1 组件特性

考察一个组件有用的方法是将其看成独立的服务供应者。当系统需要某一服务时，会调用提供相应服务的组件，无需知道此组件位于何处，也无需知道该组件是用什么程序语言开发的。例如，图书馆系统中的组件可能提供搜索功能，允许用户搜索不同图书馆的目录；从一个图形格式变换到另一个图形格式的组件（如从 TIFF 格式到 JPEG 格式）可提供数据转换功能。

将组件看成是服务提供者，强调的是可复用组件的两个关键特性：

- 1. 组件是独立可执行的实体，这是由它的接口定义的。你不必知道组件的源代码的任何信息来使用它。它可能作为一种外部服务来引用，也可能直接包含在一个程序中。
- 2. 组件所提供的服务可以通过其接口得到，而且所有的交互都是通过接口实现的。组件接口表现为参数化的过程，其内部状态是不会暴露出来的。



组件和对象

组件通常是用面向对象语言实现的。在某些情况下，对组件“提供”接口的访问是通过方法调用进行的。然而，组件和对象类是两回事。不像对象类，组件是可以独立部署的，不定义类型，是语言无关的，同时它基于标准的组件模型。

<http://www. SoftwareEngineering-9. com/ Web/ CBSE/ objects. html>

组件有两种关联接口，如图 17-2 所示。这些接口反映了组件提供的服务和组件进行正确操作所需要的服务。

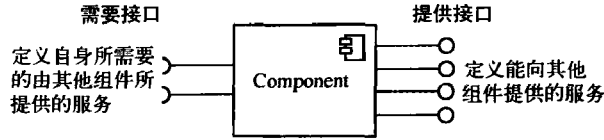


图 17-2 组件接口

1. 提供接口定义了组件所提供的服务。这个接口主要是组件的 API。它定义了组件用户可以调用的方法。在一个 UML 组件图中，组件提供接口用一个圆圈表示，圆圈在始于组件图标的一条线段的尾端。

2. 需要接口如果一个组件要进行正确的操作，指定系统其他组件必须提供哪些服务。如果这些服务不能实现组件将无法工作。这并不影响组件的独立性或部署性，因为“需要”接口没有定义如何提供这些服务。在 UML 中，一个“需要”接口的标志是用一个半圆形来表示，半圆位于始于组件图标的一条线段的尾端。注意：提供接口和需要接口图标相配如同球和球洞。

为了说明这些接口，图 17-3 是一个组件模型，这是一个收集和比较来自传感器阵列信息的组件。它过一段时间就自动收集数据，然后根据请求为调用组件提供比较后的数据。提供接口包括对传感器进行添加、移动、开始、停止和测试等方法，报告方法返回收集的传感数据，listAll 方法提供所有连接的传感器的信息。尽管在这里没有给出，但这些方法自然地有相关联的参数，如定义传感器位置等。

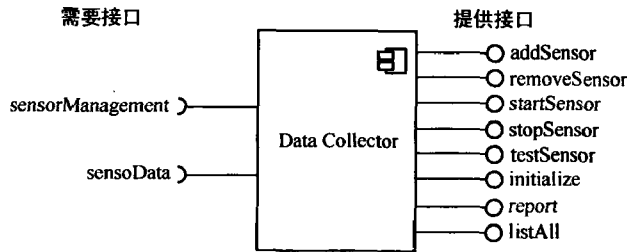


图 17-3 数据采集器组件的模型

“需要”接口是用来将组件连接到传感器上的。它假设传感器有一个数据接口（通过 sensorData 访问）和一个管理接口（通过 sensorManagement 访问）。这个接口的设计是用来连接不同类型的传感器，因而它就不能包括像 Test 和 provideReading 等这样的特殊的传感器操作。相反，一个特定类型的传感器操作使用的命令嵌入在一个字符串中，这个字符串是“需要”接口中操作的参数。适配器组件解析这个字符串并翻译成嵌入的命令到每个类型传感器的具体控制接口中。本章后面将介绍适配器的用法，以及收集器组件是怎样连接到适配器上的（见图 17-12）。

作为一项外部服务的组件和作为一个程序的元素的组件主要的不同之处是，服务是完全独立的实体。它们没有“需要”接口。不同的程序可以使用这些服务，而无需实现任何服务所需的额外支持。

组件模型

组件模型定义了组件实现、文档化及开发的标准。这些标准是为组件开发者确保组件的互操作性而设立的。它们也是为那些提供中间件的组件执行基础设施的供应商支持组件操作而设立的。目前已经提出了许多的组件模型，但是最重要的组件模型是现在的 WebServices 模型，Sun 公司的企业 Java Bean 模型 (EJB) 和微软的 .Net 模型 (Lau 和 Wang, 2007)。

Weinreich 和 Sametinger 讨论了一个理想组件模型的基本要素 (Weinreich 和 Sametinger,

2001)。图 17-4 总结了这些模型要素。该图说明, 组件模型的要素定义组件接口, 人们在程序中使用组件需知道的信息, 以及组件应该如何部署。

1. **接口** 组件是通过它们的接口来定义的。组件模型规定应如何定义组件接口及在接口定义中应该包括的要素, 如操作名、参数及异常等。模型同时需指定用于定义组件接口的语言。对于 Web 服务来说, 这是 WSDL, 将在第 19 章中讨论; EJB 是 Java 专有的, 所以 Java 可用于作为接口定义语言 (IDL); 在 .NET 中, 接口是使用一般的中间语言 (CIL) 来定义的。一些组件模型要求必须由组件定义专门的接口。这些接口用来与提供如安全性和事务管理等标准化服务的组件模型基础设施一起构成组件。

2. **使用** 为使组件远程分布和访问, 需要给组件一个特定的名字或句柄。这个必须是全球唯一的——在 EJB 中, 有一个层次化的名字, 其根是基于因特网域名的。服务有一个唯一的 URL (统一资源标识符)。

组件元数据是组件本身相关的数据, 如组件的接口和属性信息。元数据非常重要, 用户可通过元数据发现组件提供的和所要的服务。组件模型的实现通常包括访问组件的元数据的特定方法 (如 Java 中所使用的反射接口)。

组件是通用实体, 在部署的时候, 必须对组件进行配置来适应一个应用系统。例如, 图 17-3 所示的数据采集器组件需要根据传感器阵列所限定的最多数量进行定制。因此, 组件模型应该指定如何配置二进制组件使其适应特定的部署环境。

3. **部署** 组件模型包括一个描述, 此描述说明应该如何打包组件使其部署成为一个独立的可执行实体。由于组件是独立的实体, 所以它们必须与所有支持软件打包在一起, 这些软件包括组件基础设施所不提供的以及需要接口未定义的那些。部署信息包括有关包中内容的信息和它的二进制构成的信息。

不可避免的是, 当出现新的需求, 组件就必须做出改变或者被替代。因此, 组件模型应包括允许何时和怎样替换组件的控制规则。最后, 组件模型应定义应该产生的组件文档。这可以用于查找组件和决定组件是否适当。

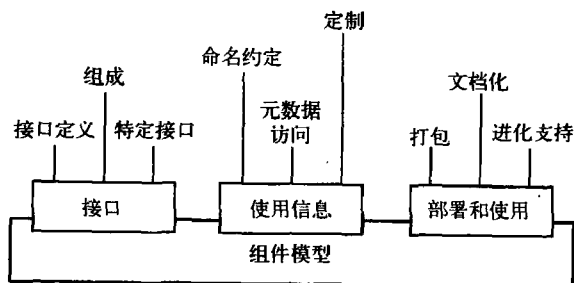


图 17-4 组件模型的基本要素

对于实现为程序单元而不是外部服务的组件来说, 该组件模型规定了必须由支持组件执行的中间件所提供的服务。Weinreich 和 Sametinger (2001) 利用操作系统类来解释组件模型。操作系统提供一组可被应用使用的通用服务。组件模型实现提供类似的共享的服务给组件。图 17-5 给出了由组件模型实现提供的某些服务。

组件模型实现提供的服务包括以下两种:

1. **平台服务**。允许组件在分布式环境下通信和互操作。在所有的基于组件的系统中, 这些是必须有的基本服务。

2. **水平服务**。这些是一般服务, 可以被很多不同的组件所使用。例如, 许多组件需要身份

验证，以确保组件服务的用户已被授权。为所有组件使用提供一组标准的中间件服务是有意义的。这些服务的可用性降低了组件开发的成本，而且意味着潜在的组件间不兼容性可以避免。

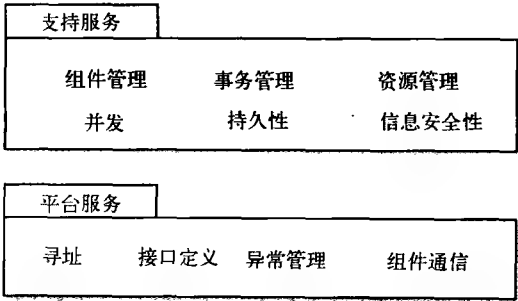


图 17-5 由组件模型提供的服务

中间件实现组件服务，并提供这些服务的接口。为利用组件模型基础设施所提供的服务，你可以认为组件被部署在一个容器中。容器是支持服务的一个实现，加上组件必须提供使之和容器整合在一起的接口定义。将组件包含在容器中意味着组件可以访问支持服务和容器可以访问组件接口。在使用时，组件接口本身不能被其他组件直接访问；它们通过一个容器接口进行访问。容器接口启动代码访问嵌入的组件的接口。

17.2 CBSE 过程

CBSE 过程是支持基于组件的软件工程的软件过程。它们考虑了复用的可能性，以及在开发和使用时可复用的组件中所涉及的不同的过程活动。图 17-6（Kotonya, 2003）给出了一个 CBSE 中过程的概览。从最高层次来说，存在两种类型的 CBSE 过程：

- 1. 面向复用的开发 这个过程是开发将被复用在其他应用程序中的组件或服务。它通常是对已存在的组件进行一般化处理。
- 2. 基于复用的开发 这个过程是复用已存在的组件和服务来开发新的应用程序的过程。

这些过程有不同的目标，因此包括不同的活动。在面向复用的开发过程中，目标是产生一个或多个可复用的组件。你必须了解你将使用的组件和你必须访问它们的源代码来将它们一般化。在基于复用的开发过程中，你不知道什么样的组件是可用的，因此你需要去发现这些组件，然后最有效地利用这些组件来设计你的系统。你可能不必访问组件的源代码。你可以从图 17-6 看到，基于复用和面向复用的基本 CBSE 过程支持那些与组件获得、组件管理和组件认证有关的过程：

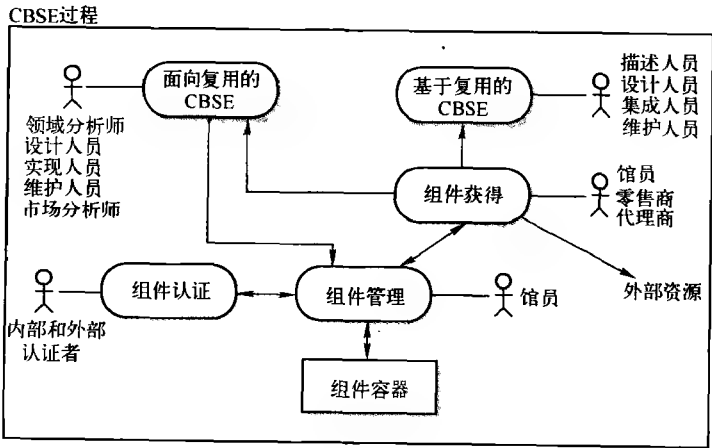


图 17-6 CBSE 过程

1. 组件获得是得到面向复用的组件或开发成一个可复用的组件的过程。它可能涉及获得内部开发的组件或服务, 或从外部资源找到这些组件。

2. 组件管理是有关管理一个企业的可复用的组件, 以确保它们得以正确分类和存储, 使面向复用成为可能。

3. 组件认证是检查组件和证实这个组件符合它的描述的过程。

一个机构所持有的组件可能会存储在组件容器中, 其中包括组件和它们的使用信息。

17.2.1 面向复用的 CBSE

面向复用的 CBSE 是开发可复用的组件和通过一个组件管理系统使对它们的复用成为可能的过程。早期的 CBSE 的支持者 (Szyperski, 2002) 的观点是要发展一个繁荣的组件市场。那将有专门的组件供应商和组件厂商, 他们根据不同的开发商来组织组件的售价。软件开发商将要购买组件来包含在系统中或是为使用他们的服务付费。然而, 这个观点并没有变成现实。目前存在相当少的组件供应商, 购买组件也是很少见的。在编写本书时, 服务市场也没有发展起来, 虽然预测在接下来的几年会有相当大扩展。

结果, 面向服务的 CBSE 最有可能发生在一个机构中, 它已允诺复用驱动的软件工程。他们希望利用在公司内不同部门中开发的软件资源。然而, 这些内部开发的组件如果不改变, 通常是不可复用的。它们经常包含具体应用程序的特征和接口, 这些在其他的组件复用的程序中可能是不需要的。

为了使组件可复用, 要改写和拓展这些组件以创建更通用的、更适合复用的版本。显然, 这有一个相关的成本问题。首先, 必须决定组件是否可能复用。其次, 未来复用节约的成本是否能抵得上使组件可复用的成本。

为回答第一个问题, 必须决定是否组件实现一个或多个稳定的领域抽象。稳定的领域抽象是应用领域中变化缓慢的基本概念。例如在银行系统中, 领域抽象包括账户、账户持有者和账目。在医院管理系统中, 领域抽象可能包括病人、治疗和护士。这些领域抽象有时称为业务对象。如果组件是对普遍使用的领域抽象或是一组相关对象的实现, 它大概就可能被复用。

为回答有关成本问题, 必须评估需要使组件可复用而需要进行的变更的成本。这些成本包括组件文档的成本、组件可靠性验证的成本及使组件更泛化的成本。可提高组件可复用性的变更包括:

- 去除那些应用特定的方法;
- 更名使其更通用;
- 添加方法提供更完备的功能覆盖;
- 为所有方法构造一致的异常处理过程;
- 添加“配置”接口允许对组件进行调整以适应不同的使用情况;
- 集成所需要的组件以增强独立性。

异常处理是一个十分困难的问题。原则上, 所有异常应该作为组件接口的一部分。组件不应该处理自身的异常, 因为每个应用程序都有自己对异常处理的需求。更确切地说, 组件应定义哪些是会产生异常并将之发布作为接口的一部分。例如, 有一个实现堆栈数据结构的简单组件, 应检测和发布栈上溢和栈下溢的异常。然而实际情况是, 这里存在两个问题:

1. 发布所有的异常将导致太多的接口, 这将更加难以理解。这可能会丢掉组件的潜在用户。
2. 组件的运行可能依靠局部异常处理, 改变它将严重影响组件的功能。

Mili 等人 (2002) 讨论了对制造可复用组件的成本及投资所能带来的利润进行估计的方法。复用组件较之重新开发组件不仅仅是生产率问题, 还包括质量收益。因为可复用组件更可靠, 且

具有市场时效性。这些额外的收益来自于更快速地部署软件。Mili 等提出各种不同公式估算这些收益，正如第 23 章中 COCOMO 模型讨论的那样（Boehm 等，2000）。然而，这些公式的参数很难准确估计，并且公式必须根据具体环境进行调整，使得使用它们很困难。作者怀疑极少有软件项目管理者使用这些模型去估计从组件可复用性的投资中得到的回报。

显然，一个组件是否可复用依赖于它的应用领域和功能性。当我们使组件具有更多的通性的时候其可复用性也在提高。然而，这也使得组件具有更多的操作，更为复杂，更难以理解和使用。

组件的可复用性与组件的可用性之间不可避免的需要折中。为了使组件可复用，你必须提供一组通用接口和操作，以满足组件的所有可能的使用方式。使组件可用，即提供简单、最小的易理解的接口。可复用性增加了复杂性，同时也降低了组件的可理解性。因而决定何时和怎样复用组件是很困难的事情。因此在设计可复用组件时，你必须找到通用性与可理解性之间的折中点。

组件的另一潜在的来源是现存的遗留系统。如第 9 章所讨论的，这些系统完成了很重要的业务功能，但却是用过时的软件技术编写的。所以很难将之与新系统一起使用。但如果将这些旧系统转化成组件，它们的功能就可以在新的应用中使用了。

当然，这些遗留系统通常没有清晰定义的需要和提供接口。为使这些组件可复用，我们必须对它进行封装，由此定义组件的接口。该封装将原代码的复杂性隐藏了起来，并为外部组件访问的服务提供了接口。虽然此封装就是一个相当复杂的软件，因为它必须访问遗留系统的功能。然而，对封装的开发成本通常远低于对遗留系统重新实现的成本。第 19 章将更加详细地介绍这个方法，并解释是如何通过服务访问一个遗留系统中的特征的。

一旦你开发和测试一个可复用的组件或服务，必须为今后的再利用来管理这个组件或服务。管理包括如何对组件进行分类，才可以发现它，使现有的组件是可用的，或者在储存库中或作为一个服务，维护有关组件的使用信息和保存不同的组件版本的记录。如果这个组件是开源的，你可以在一个公开的储存库中使它成为可用的，例如 Sourceforge。如果是计划在公司使用，你可能要使用一个内部的存储系统。

做复用项目的公司在组件成为可复用组件之前，要执行某种形式的组件认证。认证意味着除了开发者之外，可能有些人要检查这个组件的质量。在组件成为可复用组件之前，他们测试它以确保达到一个可以接受的质量标准。然而，这可能是个非常昂贵的过程，并且许多公司简单地将测试和质量检查留给组件的开发者。

17.2.2 基于复用的 CBSE

成功的组件复用需要一个适合于 CBSE 的开发过程。基于复用的 CBSE 过程必须包括搜索和整合可复用的组件。这种过程的结构在第 2 章讲过；图 17-7 给出了在 CBSE 过程中主要子活动。这个过程某些活动，如用户需求的最初发现，其执行方式与在其他软件过程中的执行方式是相同的。然而，基于复用的 CBSE 与先前的软件开发过程之间的主要不同在于：

1. 最初对用户需求的开发只需要是概要性的而不要是十分详细的，且鼓励信息持有者在定义他们的需求时能尽可能地灵活。太特殊的需求限制了能满足这种需求的组件数量。然而，不像增量式开发，我们需要完全的需求，这样我们就能尽可能多地识别出可复用的组件。
2. 在过程的早期阶段根据可利用的组件来细化和修改需求。如果可利用的组件不能满足用户需求，就应考虑相关的能被支持的需求。如果这意味着能节省开支且能快速地开发系统，或许用户愿意改变想法。
3. 在系统体系结构设计完成后，会有一个进一步的对组件搜索及设计精练的活动。一些似

乎可用的组件会变得不合适或不能与其他已选组件一起正常工作。尽管图 17-7 中没有表示出来,这意味着进一步的需求变更是必要的。

4. 开发就是将已发现的组件集成在一起的组成过程。其中包括将组件与组件模型基础设施集成在一起,有时,包括开发适配器来协调不匹配的组件的接口。当然,除了可用组件所提供的功能外还可能添加其他的功能。

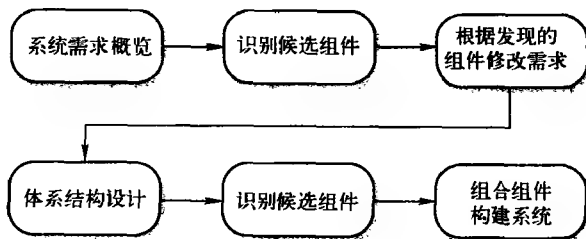


图 17-7 基于复用的 CBSE 过程

体系结构设计阶段特别重要, Jacobsen 等(1997)发现定义一个鲁棒的体系结构对于成功的复用具有决定性的意义。在这个体系结构设计阶段,你将选择一个组件模型和实现平台。然而,许多公司有一个标准的开发平台(例如.NET),因此这个组件模型是提前决定的。如第 6 章所讲过的,你也要建立系统的一个高层结构,并对系统的分布和控制做出决断。

CBSE 过程的一个独特活动是为复用识别组件或服务。这包括一系列子活动,如图 17-8 所示。首先,你的注意力集中于搜集和选择,使自己相信有可复用组件能满足需求。显然,应做些初始的检查看组件是否合适,但不需要做详细测试。在后期阶段,在系统体系结构设计完成后,应将更多的时间用于组件的有效性验证上。你需要确信所识别出的组件真正适合你的应用,若不能,就必须重复搜索和选择过程。

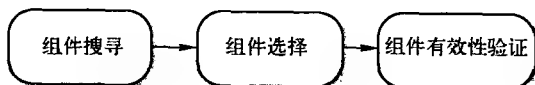


图 17-8 组件识别过程

识别组件的第一个阶段是找到内部可用的组件或从可靠的供销商那里得到的组件。正如前面提到的,有相对较少的组件供应商,因此你最有可能寻找已在自己公司开发的组件。软件开发公司可以建立他们自己的可复用组件库,而不必冒险使用外部供应商的组件。或者,你可能决定在 Web 上搜索源码库,例如 Sourceforge 或 Doogee 代码,来发现你需要的这个组件的源码是否是可用的。如果你正在搜索服务,然后有很多具体的可用的专门的网络搜索引擎,你就可以发现公开的网络服务。

一旦组件搜索过程找到可能的组件,你必须挑选出候选组件。有些情况下,这是个很明显的任务。列表上的组件直接实现用户需求,而没有竞争组件能匹配这些需求。然而另外一些情况下,选择过程是非常复杂的。组件和需求之间没有一个清晰的映射关系,你可能发现必须用多个组件才能满足一个特定的需求或一组需求。最不利的情況是,满足不同需求需要不同的一组组件,所以你必须决定哪种组件组合能最好地覆盖你的需求。

选取好系统可能包含的组件后,应对它们进行有效性验证检查它们是否像广告宣传的一样。有效性验证的程度有赖于组件来源。如果所用组件是来自熟悉的可靠的厂商,就没必要逐个测试组件性能,只需对组件做集成测试即可。相反,如果使用一个来自不熟悉的供应商的组件,在其加入系统之前必须坚持对其进行检查和测试。

组件有效性验证包括对组件开发一组测试用例（或者可能的话，拓展随组件提供的测试用例）并开发测试程序去运行组件测试。组件测试的主要问题是，组件的描述可能不够详细，不足以允许你开发一个全面的组件测试集。组件的描述总是非正规的，唯一标准的可能就是其接口描述。这样组件的信息就不足以让你开发完全的测试集合能使你确信组件的宣传的接口正是你所需要的。

在测试可复用组件是你所需要的同时，你还必须检查这个组件不包括任何恶意的代码或你不需要的功能。专业的开发者很少使用源自不可靠资源的组件，尤其是如果这些资源没有提供源代码。因此，恶意代码问题通常不会发生。然而，组件可能通常包括你不需要的功能，你必须检查这个功能是否会阻碍这个组件的使用。

不需要的功能的这个问题可能是组件本身产生的。这个可能使组件变得缓慢，导致它产生出乎意料的结果，或者在某种情况下，导致严重的系统失败。图 17-9 总结了一个在复用的系统中那些不需要的功能导致的灾难性的软件失败的情况。

阿丽亚娜 5 号 运载火箭失败

在开发阿丽亚娜 5 号空间运载火箭的时候，设计者决定复用阿丽亚娜 4 号中非常成功的惯性参照系软件。此惯性参照系软件维持火箭的稳定。他们决定不加改变地复用此软件（一般人们都会这么做），尽管它包含了额外的功能，超出了阿丽亚娜 5 号的需要。

在阿丽亚娜 5 号首次发射中，惯性导航软件在升空后失败，火箭因而失去控制。地面控制人员发送指令给火箭令其自毁，于是火箭有效载荷被摧毁。事后的调查发现导致问题的原因是某个定点数到整数转换中发生了数的溢出这是个未处理的异常。这导致了运行系统关闭了惯性参照系系统。所以火箭的稳定性得不到维持。此缺陷在阿丽亚娜 4 号中没有发生是因为它的引擎功率比较小，所转换的值不会大到溢出的程度。

缺陷发生在一段阿丽亚娜 5 号所不需要的代码上。对所复用的软件的有效性验证测试是基于阿丽亚娜 5 号的需求的。由于对失败的功能没有需求，所以没有针对它的测试。结果是在发射仿真测试中没有发现软件中的问题。

图 17-9 可复用软件验证失败的例子

阿丽亚娜 5 号运载火箭出现问题是因为对阿丽亚娜 4 号的软件所做的假设对于阿丽亚娜 5 号是不适用的。这是可复用组件的常见问题。它们是最初为某个应用环境实现的，自然地就嵌入了对那个环境的假设。这种假设很少有记录，因此，当组件被复用时，不可能导出测试来检查是否假设仍然是有效的。如果你在不同的环境中复用一个组件，你可能不会发现内含的环境假设，直到你在一个运行系统中使用这个组件。

17.3 组件合成

组件合成是指组件相互直接集成或是用专门写的“胶水代码”将它们整合在一起创造一个系统或另一个组件的过程。合成组件有很多种不同的方法，正如图 17-10 中描述的那样。从左到右，图中分别是顺序合成、层次合成和叠加合成。在下面的讨论中，假设使用两个组件（A 和 B）来合成一个新的组件：

1. 顺序合成是对应于图 17-10a 中的情形。你可以用两个已经存在的组件来创造一个新的组件，通过按顺序调用已经存在的组件。你可以把这个合成看做是“提供接口”的合成。也就是说，调用 A 提供的服务，然后用 A 返回的结果调用组件 B 提供的服务。在顺序合成中，组件间不会相互调用。需要特定的胶水代码来按照正确的顺序调用组件服务以及确保组件 A 提供的结果和组件 B 所期望的输入相兼容。组合的“提供”接口依赖于 A 和 B 的合并功能，但通常不是它们的“提供接口”的一个简单合成。这种组合的类型可能适用于作为程序元素的组件或是作为服务的组件。

2. 层次合成是对应于图 17-10b 中的情形。这种情况发生在一个组件直接调用由另一个组件所提供的服务时。被调用的组件为调用的组件提供所需要的服务。因此，被调用组件的“提供”接口必须和调用组件的“需要”接口兼容。组件 A 直接调用组件 B，如果它们的接口相匹配，则组件 A 可以直接调用组件 B，这就不需要额外的代码。然而，如果组件 A 的“需要”接口和组件 B 的“提供”接口不匹配，则需要一些转换代码。因为服务没有“需要”接口，当组件作为网络服务来实现时，这个合成模式是不能使用的。

3. 叠加合成是对应于图 17-10c 中的情形。这种情况发生在两个或两个以上组件接合在一起（叠加）创建一个新组件的时候，这个新组件合并了它们的各自功能。这个新组件的“提供”接口和“需要”接口是相应的组件 A 和组件 B 的所各自对应的接口的一个捆绑。通过合成组件的外部接口来分别调用组件 A 和 B。A 和 B 不相互依赖，也不相互调用。这种合成类型适合于组件是程序单元或者组件是服务的情形。

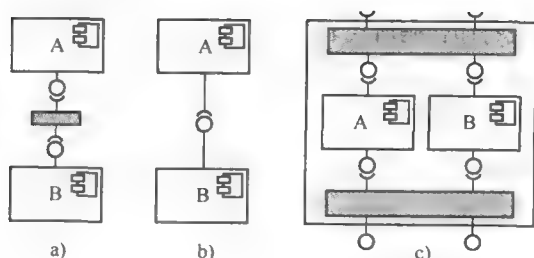


图 17-10 组件合成类型

当创建一个系统时，可能用到所有形式的组件合成方式。对所有情况，你都必须写“胶水代码”来连接组件。例如：在顺序合成中，组件 A 的输出成为组件 B 的输入。这就需要有一个中间声明来调用组件 A，收集结果然后用该结果作为参数来调用组件 B。当一个组件调用另外一个组件时，你可能需要引入一个过渡组件以确保“提供”接口和“需要”接口是兼容的。

当你编写组件尤其是为了合成来写组件时，必须注意设计好组件接口以使其在这个系统中相互匹配。这样我们就可以很容易地把这些组件合成为一个单元。然而，当独立开发可复用组件时，你经常会面临接口不兼容的问题，即我们所要组合的组件的接口不一致的问题。有 3 种类型不兼容会发生：

1. **参数不兼容** 接口每一侧的操作有相同的名字，但参数类型或参数数目是不同的。
2. **操作不兼容** 提供接口和需要接口的操作名不同。
3. **操作不完备** 一个组件的提供接口是另一个组件需要接口的一个子集，或者相反。

对所有情况，都必须通过编写适配器组件来解决不兼容的问题，适配器组件使两个可复用组件的接口相一致。适配器组件将一个的接口转换为另外一个的接口。适配器的准确形式依赖于合成的类型。例如，在下个例子中，适配器仅是从某个组件得到结果，然后将其转化为另一组件的输入。在其他情况，适配器可能作为组件 B 的代理被组件 A 调用。这种情况发生在如果 A 希望去调用 B，但是 A 的“需要”接口细节和 B 的“提供”接口细节不匹配的情形。适配器通过将来自 A 的作为输入的参数转换为 B 需要的输入参数来协调这些差异。然后就可调用 B，将服务传达给 A 了。

为了说明适配器，考虑图 17-11 中的组件，它们的接口是不兼容的。这可能是应急服务中所使用的系统的一部分。当应急操作员接听呼叫时，电话号码被输入到 addressFinder 组件来定位地址。然后，用 mapper 组件打印出地图发给分派到应急现场的车辆。事实上，组件的接口比这里给出的要复杂得多，但是这个简化的版本有助于说清楚适配器的概念。

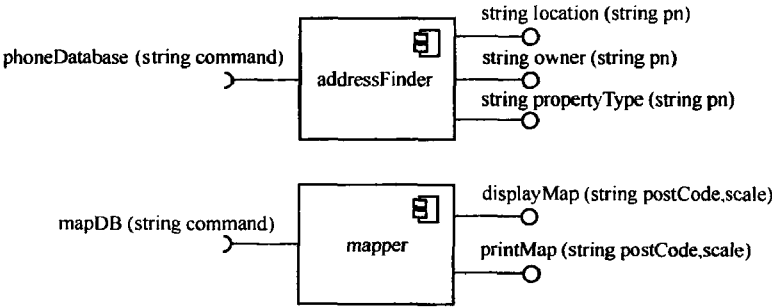


图 17-11 带有不兼容接口的组件

第一个组件，addressFinder，查找与电话号码相匹配的地址。它还能返回与电话号码相应的房产主人。mapper 组件使用邮政编码（在美国，为标准 ZIP 码，附加 4 位标示房产位置的阿拉伯数字），以某个比例尺显示或打印代码所代表的周围相关地区的街区地图。

原则上这些组件是兼容的，因为房产位置包括邮编或 ZIP 码。然而，你必须写一个叫做 post-CodeStripper 的适配器组件，它从 addressFinder 得到位置数据并剥离出邮政编码。这一邮政编码转而作为 mapper 的输入，街道地图以 1:10000 的比例显示。下面的代码说明了实现它所需要的调用序列：

```
address = addressFinder.location (phonenumber) ;
postCode = postCodeStripper.getPostCode (address) ;
mapper.displayMap(postCode, 10000) ;
```

使用适配器组件的另一种情况是在层次合成中。即某一组件需利用另一组件，而两者的“提供”接口和“需要”接口是不兼容。图 17-12 说明了适配器的用法，这里一个适配器是用来链接数据采集器和传感器。它们会用在实现一个野外气象站系统中，正如第 7 章讨论的。

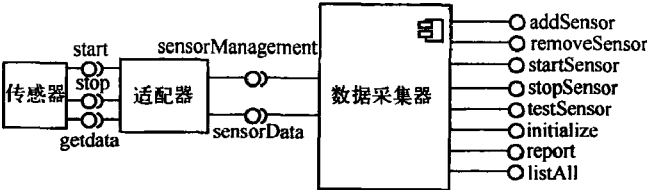


图 17-12 链接数据采集器和传感器的适配器

传感器组件和数据采集器组件使用适配器合成。适配器使数据采集器组件的“需要”接口与传感器组件的“提供”接口相一致。数据采集器组件在设计上是采用一种通用的“需要”接口，这个接口支持传感器数据收集和传感器管理。对每一个这些操作来说，它的参数是代表具体传感器指令的文本字符串。例如，发布一个收集命令，你必须写 sensorData（“collect”）。正如图 17-12 中所表示的，传感器本身有单独的操作诸如“start”、“stop”和“getdata”。

适配器解析输入串，识别命令（例如，收集），然后调用 Sensor.getdata 来得到传感器的值。然后它将结果（作为字符串）返回给数据收集器组件。这个接口的风格意味着数据收集器可以和不同类型的传感器相互作用。对于每个类型的传感器要实现一个单独的适配器，它可以将传感器指令从 Data Collector 转换到实际的传感器接口。

前面关于组件合成的讨论假设你可以从组件文档中判断出接口是否兼容。当然，接口的定义包括操作名和参数类型，因此可根据这些评估兼容性。然而，却不能够依赖组件文档来判断出接口是否是语义兼容的。

为了说明这个问题，考虑图 17-13 中给出的合成例子。这些组件是用来实现一个能从数码相机传输图像文件并将它存储在图片库中的系统。系统用户可以提供额外的信息来描述图片和对图片制作目录。为避免混乱，这里并没给出所有的接口方法，只简单给出这些方法，因为需要它们来解释组件文档化问题。Photo Library 接口方法是：

```
public void addItem (Identifier pid ; Photograph p; CatalogEntry
photodesc) ;
public Photograph retrieve (Identifier pid) ;
public CatalogEntry catEntry (Identifier pid);
```

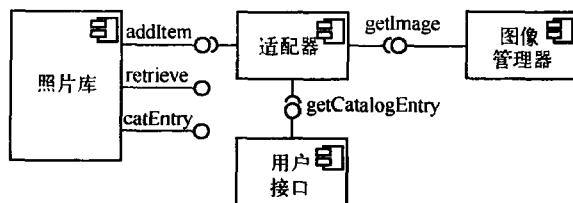


图 17-13 图片库组成

假设在 Photo Library 中的 addItem 方法的文档如下：

此方法将图片加入图片库，为图片起一个名字并给一个目录。

此描述看起来是解释组件做什么，但是让我们考虑如下问题：

- 若图片名字已与库中其他图片名重复会怎样？
- 图片描述符在目录入口和图片本身都有吗？也就是说，如果删除照片，是否要删除目录信息？

在 addItem 的非正式描述中是没有足够的信息回答这些问题的。当然，是可以添加更多的信息到方法的自然语言描述中，但是，总的来说，解决这种二义性的最好的方法是使用形式化语言来描述接口。图 17-14 所给出的描述是 Photo Library 的接口描述的一部分，它是对非形式化描述添加了一些信息。

```

- The context keyword names the component to which the conditions apply
context addItem

- The preconditions specify what must be true before execution of addItem
pre:   PhotoLibrary.libSize() > 0
       PhotoLibrary.retrieve(pid) = null

- The postconditions specify what is true after execution
post:  libSize() = libSize()@pre + 1
       PhotoLibrary.retrieve(pid) = p
       PhotoLibrary.catEntry(pid) = photodesc

context delete
pre:   PhotoLibrary.retrieve(pid) <>null ;
post:  PhotoLibrary.retrieve(pid) = null
       PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
       PhotoLibrary.libSize() = libSize()@pre-1

```

图 17-14 Photo Library 接口的 OCL 描述

图 17-14 中的描述使用了前置条件和后置条件，这是定义在一个基于对象约束语言（OCL）上的一套符号，而 OCL 是 UML 的一部分（Warmer 和 Kleppe, 1998）。OCL 是设计用来描述 UML 对象模型中的约束的；它允许你去表达必为真的谓词，或者是在执行某个方法之前必为真的谓词，或者是在执行完某个方法后必为真的谓词。这些都是不变量，前置条件和后置条件。为在操

作前访问一个变量的值，可以在它的名字后加上@ Pre。因此，使用 age 作为一个例子：

```
age = age@pre + 1
```

此语句意味着 age 的值在操作后要比其在操作前的值多 1。

基于 OCL 的方法越来越多地使用在为 UML 模型加入语义信息，OCL 描述在模型驱动工程中可用于驱动代码生成器。一般方法得自 Meyer 采用 Contract 方法的设计（Meyer, 1992），在所谓的 Contract 方法中，通信的对象的接口和行为规范是形式化定义的，并且被运行时系统执行。Meyer 认为，如果我们要开发可靠的组件，那么用 Contract 的设计就是绝对必要的。

图 17-14 包括了 Photo Library 中 addItem 方法和 delete 方法的描述。所定义的方法是通过关键词 Context 来指示的，前置条件和后置条件是用 pre 和 post 来指示的。addItem 的前置条件陈述为：

- 1. 图片库中的图片不能有和待加入的图片有相同的标识。
- 2. 库必须是存在的——假定创建一个库就加入一个项在其中，这样库的大小始终大于 0。
- 3. addItem 的后置条件声明为：
库的大小增 1（因此只产生一个入口）。

如果用同一个标识符去取图片，就可以得到新添加的那个图片。
如果再次用相同的标识符在目录中查找，就可以得到所生成的目录入口。

Delete 的描述提供更多的信息。前置条件陈述为：要删除一个项，它必须在库中，在删除照片后不能再得到，且库的大小要减 1。然而，delete 并不删掉目录入口——你仍在照片被删除后可以再次得到它。其原因在于你可能希望在目录中维持关于照片为何被删除、它的新位置等信息。

当你通过合成组件来创建一个系统时，你会发现在功能性需求和非功能性需求之间，在尽可能快速移交软件和创建一个能随需求变更而进化的系统之间，都存在着潜在的冲突。你将必须做折中的地方有：

- 1. 在交付系统功能性需求方面哪种组件的合成方式是最有效的？
- 2. 什么样的组件合成将可以使得合成组件在需求变化时更容易调整。
- 3. 合成系统将有那些总体特性？这些总体特性是像性能和可靠性等特性。只有当整个系统实现后你才能够评估这些特性。

不幸的是，解决合成问题的方案在许多情况下可能是相互冲突的。例如，考虑如图 17-15 中所描述的情况，系统可用两种合成而创建。系统为数据采集和报告系统，数据从不同的数据源采集而来，存储到一个数据库，然后可以产生不同汇总数据的报告。

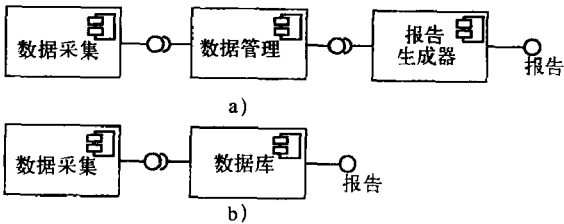


图 17-15 数据采集和报告生成组件

这里，在适应性和性能两者之间是有潜在冲突的。图 17-15a 所示合成方式适应性更强，而图 17-15b 所示合成方式也许更快和更可信。图 17-15a 所示合成方式的优势在于报告和数据处理是分离的，因此对未来的变更更为灵活。数据管理系统可以被替换，且如果需要报告，而现有的报告系统不能生成，组件同样可被替换而不必改变数据管理组件。

图 17-15b 所示合成方式中，使用的是内嵌报告设施（如微软的 Access）的数据库组件，该合成方式的主要优点在于组件很少，因此就可能很快实现，因为没有组件间太多的通信负荷。此

外，应用于数据库的数据整体性规则同样也应用于报告。这些报告不可能以不正确的方式组合数据。在图 17-15a 所示合成方式中没有这种限制，因此在报告中就更容易出错。

总的说来，一种好的合成原则是对关注焦点分离的原则。也就是说，你应试图用这样的方式设计你的系统，即每个组件都有清晰定义的任务，理想情况是，这些任务彼此不会重叠。然而，买一个多功能的组件会比买两个或三个独立组件更便宜。而且，使用多个组件在可靠性或在性能上可能会有损失。

要点

- 基于组件软件工程是一种基于复用方法来定义、实现和组合松散耦合的独立组件使之成为一个系统。
- 组件是一个软件单元，它的功能和可依赖性完全由一套公共接口定义。组件可以与其他组件相组合而无需考虑它们的实现细节，而且可以部署为一个可执行单元。
- 组件可能实现为程序单元将被包含在系统当中，或是实现为在系统中引用的外部服务。
- 组件模型定义了一组组件标准，包括接口标准、使用标准和部署标准。组件模型的实现提供了一套水平服务，可以为所有组件所用。
- 在 CBSE 过程中，我们将不得不将需求工程过程与系统设计交织在一起。我们不得不在所希望的需求和从现存的可复用组件能得到的服务两者之间采取折中。
- 组件合成是将组件“捆”到一起来创建一个系统的过程。合成的类型包括顺序合成、层次合成及叠加合成。
- 当合成不是专为你的应用写的可复用组件时，通常要编写适配器或“胶水代码”以使不同组件接口互相兼容。
- 当选择合成方式时，必须考虑系统所需的功能，非功能性需求和当系统发生改变时，一个组件能被另一组件代替的难易。

进一步阅读材料

《Component-based Software Engineering: Putting the Pieces together》这是一本论文集，收集了不同作者关于 CBSE 不同方面的文章。与其他文集一样，它相当的杂，但比 Szyperski 的书来说，在基于组件的软件工程的一般问题上覆盖面更广（G. T. Heineman and W. T. Councill, Addison-Wesley, 2001）。

《Component Software: Beyond Object-Oriented Programming, 2nd ed》这是本书的第 2 版，此书是讨论 CBSE 中的技术和非技术问题的第一部著作。它包括了比 Heineman 和 Councill 的书更专业的技术的细节的讨论，以及包括了对市场问题的透彻讨论（C. Szyperski, Addison-Wesley, 2002）。

《Specification, implementation and deployment of components》这是一篇非常好的关于 CBSE 基本教程的读物。CACM 的同一期中包括组件和基于组件开发的论文（I. Crnkovic, B. Hnich, T. Jonsson and Z. kiziltan, Comm. ACM, 45(10), October 2002）。<http://dx.doi.org/10.1145/570907.570928>。

《Software Component Models》这是一篇对商业和研究用的组件模型的一个全面的讨论，它将这些模型进行分类，并解释了它们之间的差异（K-K. Lau and Z. Wang, IEEE Transactions on Software Engineering, 33 (10), October 2007）。

<http://dx.doi.org/10.1109/TSE.2007.70726>。

练习

17.1 为什么对所有组件的交互都定义为通过“需要”接口和“提供”接口是重要的？

- 17.2 组件独立原则表示用完全不同的方法实现的一个组件代替另一个组件是有可能的。举例说明这样的组件代换可能会出现不希望的后果,且会导致系统失败。
- 17.3 组件作为程序元素与作为服务之间有什么根本不同?
- 17.4 为什么组件应基于标准组件模型是很重要的?
- 17.5 列举一个实现抽象数据类型(如堆栈或链表)的组件的例子,说明为什么通常可复用组件总是需要扩展和改写?
- 17.6 说明为什么没有组件源代码,实现可复用组件的有效性验证是相当困难的。以怎样的形式化组件描述才能简化验证问题?
- 17.7 设计一个可复用组件的“提供”接口和“需要”接口,这个组件在 MHC-PMS 中可能用于表示一个病人。
- 17.8 举例说明需要不同类型的适配器来支持顺序合成、层次合成和叠加合成。
- 17.9 设计组件接口,该组件可能用于应急控制室的系统中。你应为呼叫记录组件设计接口,该组件记录所发生的呼叫,再为车辆发现组件设计接口,该组件在给定邮政编码(邮政编码)和事故类型后,找出最近的可用车辆来分派到出事现场。
- 17.10 有人提出应建立独立的认证机构。供应商可以提交他们的组件给该机构,由此机构来验证它的可信性。这种认证机构的优点和缺点是什么?

参考书目

Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. and Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ.: Prentice Hall.

Councill, W. T. and Heineman, G. T. (2001). 'Definition of a Software Component and its Elements'. In *Component-based Software Engineering*. Heineman, G. T. and Councill, W. T. (ed.). Boston: Addison-Wesley, 5-20.

Jacobson, I., Griss, M. and Jonsson, P. (1997). *Software Reuse*. Reading, Mass.: Addison-Wesley.

Kotonya, G. (2003). 'The CBSE Process: Issues and Future Visions'. *Proc. 2nd CBSEnet workshop*, Budapest, Hungary.

Lau, K.-K. and Wang, Z. (2007). 'Software Component Models'. *IEEE Trans. on Software Eng.*, 33 (10), 709-24.

Meyer, B. (1992). 'Design by Contract'. *IEEE Computer*, 25 (10), 40-51.

Meyer, B. (2003). 'The Grand Challenge of Trusted Components'. *ICSE 25: Int. Conf. on Software Eng.*, Portland, Oregon: IEEE Press.

Mili, H., Mili, A., Yacoub, S. and Addy, E. (2002). *Reuse-based Software Engineering*. New York: John Wiley & Sons.

Pope, A. (1997). *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*. Harlow, UK: Addison-Wesley.

Szyperski, C. (2002). *Component Software: Beyond Object-oriented Programming*, 2nd ed. Harlow, UK: Addison-Wesley.

Warmer, J. and Kleppe, A. (2003). *The Object Constraint Language: Getting your models ready for MDA*. Boston: Addison-Wesley.

Weinreich, R. and Sametinger, J. (2001). 'Component Models and Component Services: Concepts and Principles'. In *Component-Based Software Engineering*. Heineman, G. T. and Councill, W. T. (ed.). Boston: Addison-Wesley, 33-48.

分布式软件工程

目标

本章的目标是介绍分布式系统工程和分布式系统体系结构，读完本章，你将了解以下内容：

- 在设计和实现一个分布式软件系统时不得不考虑的关键问题；
- 客户机/服务器计算模型和客户机/服务器系统的分层体系结构；
- 分布式系统体系结构常用到的模式，对于不同类型的系统所适合的体系结构；
- 理解软件作为一种服务的概念，提供了对远程应用程序的基于 Web 的访问。

事实上，所有大型计算机系统现在都是分布式系统。分布式系统与所有的系统组件在一台单独的计算机上执行的集中式系统相反，在分布式系统中包含了许多计算机。Tanenbaum 和 Van Steen (2007) 对分布式系统做了定义：

“……一个独立计算机的集合体，给用户的感觉却是面对一个独立的系统。”

很明显，分布式系统工程与其他软件工程有许多相同的地方，但也有许多特殊问题需要在设计这类系统时予以考虑。这些问题是由于系统组件可能运行在独立管理的计算机上以及组件间要经过网络通信所引起的。

Coulouris 等 (2005) 找出了用分布式方法开发系统的如下优势：

1. **资源共享** 分布式系统允许硬件、软件资源，如磁盘、打印机、文件和编译器等共享使用。显然，在多用户系统上也是可以共享资源的，只不过在这种情况下资源必须放在一个中央计算机上统一管理。
2. **开放性** 是指系统通过添加非私有资源来扩展自己的能力。分布式系统是开放的系统，它一般包括来自不同厂商的软件和硬件的兼容产品。
3. **并发性** 在分布式系统中，有许多过程可以在网络的不同计算机上同时运行。这些过程在其正常运行期间可以（但不是必需）彼此通信。
4. **可扩展性** 原则上，分布式系统至少要有可扩展性，系统的能力可以通过增加新的资源来满足对系统的新的需求。实际过程中，可扩展性可能受到网络的限制。
5. **容错性** 多台计算机及其信息复制能力意味着系统对硬件和软件错误具有相当的容错能力（见第 13 章）。在绝大多数分布式系统中，当有错误发生时，可能会使提供的服务能力下降，但彻底丧失服务能力只有在发生网络错误时才会出现。

对于大型系统，这些优点意味着分布式系统已经大量地替换了像在 20 世纪 90 年代中所开发的大型机遗留系统。然而，还有许多个人计算机应用系统（例如，照片编辑系统）不是分布式的，这些系统运行在单个的计算机系统上。许多嵌入式系统同样也是单处理器系统。

分布式系统本质上比集中式系统更复杂。这使得分布式系统更难设计、实现和测试。由于系统组件和系统基础结构之间交互的复杂性，分布式系统的总体特性是很难了解的。举例来说，系统性能不再取决于一个处理器的运行速度，它依赖网络带宽、网络负载和系统中所有计算机的速度。将系统某部分的资源转移到另外地方将严重影响系统的性能。

此外，正如 WWW 的所有用户所了解的，分布式系统的响应是不可预知的。这个响应依赖于系统的总的负荷、它的体系结构以及网络负荷。由于这些因素都是在一个较短时间内变化的，

同一个用户的两个请求之间就会有极大的响应差别。

在过去的几年里影响分布式软件系统最重要的进展是面向服务的出现。本章大部分的重点是分布式系统的一般问题，但是，18.4节会涉及应用作为服务来部署的概念，这与第19章的内容是呼应的。第19章的重点是服务（作为面向服务体系结构的组件）以及面向服务软件工程的一般问题。

18.1 分布式系统的问题

正如本章的引言中提到的，分布式系统比在单处理器上运行的系统要复杂得多。这种复杂源于分布式系统不可能有一个自顶向下的控制模型。系统中提供功能的节点通常是独立的系统，它们没有单独管理系统的权限。连接这些节点的网络是一个单独管理的一个系统。这个系统本身是很复杂的，并且不能被使用网络的用户所控制。因此，分布式系统的运行本质上就会有不可预测性，系统的设计者必须要考虑到这一点。

在分布式系统工程中一些不得不考虑的重要的设计问题是：

1. **透明性** 分布式系统在多大程度上应该给用户呈现为一个单个系统？对于用户何时了解系统的分布性是有益的？
2. **开放性** 系统是否应该使用支持互操作性的标准协议来设计？是否应该使用更多的专业的协议来限制设计者的自由度？
3. **可扩展性** 系统应如何构建才能可扩展？也就是，整个系统如何来设计才能提升当系统需求增加时的反应能力？
4. **信息安全性** 如何定义和实现可用的信息安全策略，并应用到一群独立管理的系统？
5. **服务质量** 交付给系统用户的服务质量应如何定义？系统应如何实现才能给所有的用户一个可接受的服务质量？
6. **失败管理** 系统失败如何检测、抑制（使对系统的其他组件造成的影响最小）和修复？



CORBA——通用对象请求代理体系结构

CORBA 是一个闻名的对中间件系统的描述，它是由对象管理集团（OMG）于20世纪90年代提出来的。它的目标是建立一个开放标准允许开发中间件支持分布的组件通信和执行，也提供一系列被这些组件使用的标准服务。

已经有了 CORBA 的多个实现，但是系统从未达到过临界规模。用户倾向于使用专用的系统或是投向面向服务的体系结构。

<http://www.SoftwareEngineering-9.com/Web/DistribSys/Corba.html>

理想情况下，系统是分布的对于用户应该是透明的。这意味着用户会把系统看成一个单一的系统，这个单一的系统行为不会受分布的影响。事实上，这是不可能完成的。因为，分布式系统的集中控制是不可能的，并且系统中单独的计算机在不同时间的行为是不同的。此外，由于信号在网络中的传输需要一定的时间长度，网络延迟是不可避免的。延迟的长度依赖于系统中资源的位置、用户网络连接的质量和网络负载。

实现透明性的设计方法依赖于在分布式系统中创建资源的抽象，这样，这些资源的物理实现就可以改变而无需应用系统做任何改变。中间件（在18.1.2节中将讨论到）用于将程序中引用的逻辑资源映射到实际的物理资源上，及管理这些资源之间的交互。

实际上，一个系统完全透明是不可能的，通常，用户会意识到他们正在与一个分布式系统打

交道。因此，你可能会决定最好是向用户显示这是个分布式系统。那么，他们就可以为分布式系统所可能带来的后果做准备，比如网络延迟、远程节点故障等。

开放性的分布式系统是依据普遍接受的标准来建立的系统。这意味着提供商提供的组件可以整合到系统中而且可以和其他系统组件互操作。在网络层面，开放性现在理所当然地被认为要遵从互联网协议，但是在组件层面，开放性还没有普及。开放性意味着系统组件可以使用任何编程语言独立的开发，并且如果这些组件符合标准，这些组件将会和其他组件一起运作。

20 世纪 90 年代的 CORBA (Pope, 1997) 标准制定，旨在实现开放性但是从未达到临界规模。相反，许多公司选择使用 SUN 公司和微软公司私有的组件标准来开发系统。这些标准提供了更好的软件实现和软件支持以及对产业协议更好的长期支持。

面向服务体系结构的 Web 服务标准（在第 19 章中讨论）的进展有可能成为开放的标准。然而，这些标准有很大的阻力因为其效率低下。一些面向服务系统的开发者选择被称作 RESTful 的协议来作为替代品，因为这些协议本身要比 Web 服务协议的开销低很多。

系统的可扩展性反映了系统能在需求增长的情况下提供高质量的服务的能力。Neuman (1994) 识别出可扩展性的 3 个方面：

1. **规模** 应该可以增加更多的资源到系统来应付越来越多的用户。
2. **分布** 应该可以地域性地分散系统的组件而不会降低系统的性能。
3. **可管理性** 当系统的规模增加时应该可以管理它，即使系统的组件位于独立的机构。

所谓的规模，有增强扩展和增加扩展的区别。增强扩展意味着用更强大的资源替换系统中的资源。例如，你或许会把服务器的内存由 16 GB 增加到 64 GB。增加扩展是指向系统增加更多的资源（例如，增加一个额外的服务器与现存的服务器一起工作）。增加扩展通常要比增强扩展更有成本效益，但是这意味着系统要设计得能并行处理才行。

在本书的第二部分中，已经讨论了一般的信息安全问题和信息安全工程的问题。但是，与集中式系统相比，分布式系统可能遭到攻击的方式就会明显增加。如果系统的一个部件被成功的攻击，那么攻击者很可能以此作为“后门”来入侵系统的其他部件。

分布式系统必须防卫以免遭到以下类型的攻击：

1. **拦截**，系统部件之间的通信被攻击者拦截，因此将会丧失机密性。
2. **中断**，系统的服务遭到攻击并且不能按照预期来提供服务。拒绝服务攻击包括使用非法请求轰击一个节点，使得该节点不能处理合法的请求。
3. **更改**，系统中的服务和数据被攻击者修改。
4. **捏造**，攻击者生成本不应该存在的信息并且使用这些信息获得一些权限。例如，一个攻击者可能会生成一个不真实的密码入口并借此访问系统。

分布式系统最大的难点是建立一个能可靠地应用于系统中所有组件的信息安全策略。如第 11 章所述，一个信息安全策略规定了一个系统所能达到的安全级别。信息安全机制，比如加密和认证，被用来执行信息安全策略。在分布式系统中出现的难点是由于不同的机构可能拥有系统的组件。这些机构或许有互不兼容的信息安全策略和信息安全机制。不得不制定安全协商以允许系统一起工作。

分布式系统提供的服务质量 (QoS) 反映了系统的一种能力，即可靠地提供服务并使得响应时间和吞吐量对于用户来说都是可接受的。理想情况下，服务质量需求应该事先具体化并且设计和配置系统来提供这样的服务质量。不幸的是，这通常是不现实的，有两个原因：

1. **设计和配置系统提供高负荷下的高服务质量是不符合成本效益的。**这可能因为提供的资源在大多数时间是闲置的。“云计算”主要争论之一是它部分地解决这个问题。使用云，当需求增加时很容易增加资源。

2. 服务质量参数可能会相互矛盾。例如，可靠性的提升可能意味着吞吐量的减少，这是由于引进了检查程序来保证所有的系统输入都是合法的。

当系统处理时间紧迫的数据如语音或视频流时，服务质量是至关重要的。在这种情况下，如果服务质量降到了阈值以下，那么语音或视频可能会衰减到无法接受。系统处理语音和视频应该包括服务质量协商和组件管理。这些应该评估服务质量需求而不是可用资源，如果这些是不足的，协商提供更多的资源或者降低服务质量目标。

在分布式系统中出现失败是不可避免的，所以系统在设计上必须要适应这些失败。失败是无处不在的，以至于 Leslie Lamport，分布式系统的著名研究者，对分布式系统提出了一个著名的定义：

“在你从未听说过的一个系统崩溃阻止你做任何工作的时候，你才知道你有一个分布式系统。”

失败管理包括应用容错技术（在第 13 章提到）。分布式系统因而应该包括一个发现机制，即一旦发现系统的一个组件已经失败，要持续尽可能地提供很多服务而不管组件失败，以及尽可能自动地从故障中恢复。

18.1.1 交互模型

分布计算系统中的计算机之间可能会发生两种基本类型的交互：过程式交互和基于消息的交互。过程式交互指的是一台计算机请求其他计算机提供的一个已知的服务并（总是）等待将要传送的服务。基于消息的交互指的是“发送”计算机在消息中定义所需要的信息并发送给另一台计算机。较之对另一台机器的过程调用，消息总是在一个单独的交互中传送更多的信息。

为了说明过程式交互和基于消息交互的区别，我们考虑在餐馆中点菜的情形。当你和服务员交谈时，你是在一系列同步的过程式交互中指定你要的东西的。你做出了请求；服务员收到请求；你做出另一以请求，请求收到；等等。这好比软件系统中的组件交互，其中一个组件调用来自其他组件的方法。服务员记下你所点的菜，以及和你一起的其他人所点的菜，然后服务员将包括了所点的每个菜的细节的消息传递给厨房以准备食物。本质上讲，服务员传递消息给厨房，消息定义了需要准备的食物。这是基于消息的交互。

图 18-1 说明了如同一系列调用的同步点菜的过程，而图 18-2 给出了一个假设的 XML 消息，它定义了一桌 3 人所点的菜。这之间的信息交流方式的差异是很明显的。服务员所记录的是一系列的交互，每一个交互定义了所点的一部分菜。而服务员和厨房之间有一个单独交互，所传递的消息定义了完整的点菜内容。

在分布式系统中过程式通信往往是通过远程过程调用（RPC）实现的。在远程过程调用过程中，一个组件调用另一个组件就像是调用本地的程序或方法。系统中的中间件拦截调用并把它传送给一个远程的组件。远程组件执行了所要的计算，通过中间件返回结果给调用组件。Java 中，远程方法调用（RMI）与远程过程调用是差不多的，但不完全相同。RMI 框架在 Java 程序中处理对远程方法的调用。

RPC 要求一个“桩”以便所要调用的过程在请求的计算机上是可访问的。这个桩被调用并把过程的参数转化为一种标准的表示以便传送给远程程序。经过中间件，接着发送执行请求给远程程序。远程过程使用库函数转换参数成需要的格式，执行相应计算，并传输结果给中间件以便继续传输给发送者。

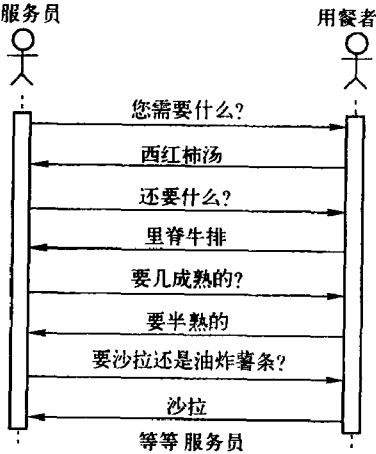


图 18-1 用餐者和服务员之间的过程式交互

```

<starter>
  <dish name = "soup" type = "tomato" />
  <dish name = "soup" type = "fish" />
  <dish name = "pigeon salad" />
</starter>
<main course>
  <dish name = "steak" type = "sirloin" cooking = "medium" />
  <dish name = "steak" type = "fillet" cooking = "rare" />
  <dish name = "sea bass">
</main>
<accompaniment>
  <dish name = "french fries" portions = "2" />
  <dish name = "salad" portions = "1" />
</accompaniment>

```

图 18-2 服务员和厨房员工之间的基于消息的交互

基于消息的交互通常是组件创建一个消息，这个消息详细地说明了来自另一个组件的服务需求。通过中间件，消息发送给接收组件。接收者分析消息，执行计算，并为发送组件创建一个带有需求结果的消息。这个消息接着传到中间件以便传送给发送组件。

远程过程调用方法带来的一个问题是调用者和被调用者需要在通信时都是有效的，它们必须知道如何相互指引。本质上，远程过程调用和本地程序和方法调用有着同样的需求。相反，在基于消息的方法中，不可用是可以被容忍的，因为消息仅仅呆在队列里直到接收者变为可用。此外，消息的发送者和接收者没必要知道彼此。他们只是与中间件通信，中间件负责确定消息传送到合适的系统中。

18.1.2 中间件

在分布式系统中，不同的组件可能用不同的程序语言来实现，且这些组件可能运行在不同类型的处理器上。数据模型、信息表示法以及通信协议可能都不一样。因此，分布式系统就需要某种软件来管理这些不同部分，确保它们能通信和交换数据。

中间件这个术语指的就是这样一种软件，它位于系统的不同分布式组件之间。图 18-3 说明了中间件是操作系统和应用程序中间的一层。中间件通常实现为一系列库，这些库被安装到每一个分布的计算机上，加上一个运行时系统管理通信。

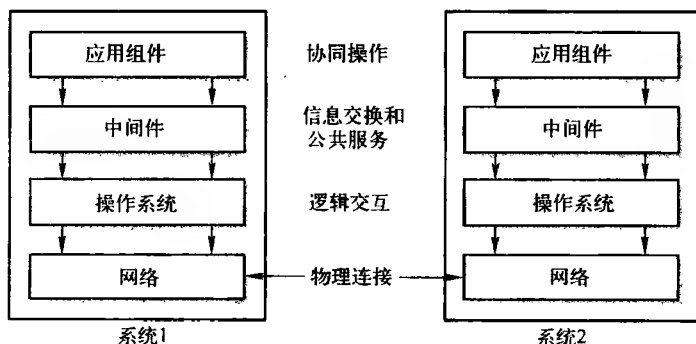


图 18-3 分布式系统中的中间件

Bernstein (1996) 总结了支持分布式计算的中间件的不同类型。中间件是一种通用软件，通常中间件不是由应用开发人员编写的，而是购买现成的。中间件的例子有负责数据库通信管理的软件、事务管理器、数据转换器和通信控制器等。

在一个分布式系统中，中间件通常提供两种不同类型的支持：

1. 交互支持，中间件协调系统中的不同组件之间的交互。中间件提供位置透明性，因为组件不需要知道其他组件的物理位置。如果使用不同的程序语言来实现组件、事件检测和通信等时，那么中间件还可能会支持参数转换。
2. 提供公共服务，即中间件提供对服务的可复用的实现，这些服务可能会被分布式系统中的很多组件所需要。通过使用公共服务，组件可以很容易地相互协作并且可以持续地向用户提供服务。

18.1.1 节已经给出了中间件可以提供的交互支持的例子。你可以使用中间件来支持远程过程和远程方法调用和消息交换等。

公共服务是指被不同组件需求的服务，不管这些组件的功能是什么。如第 17 章中所述，这些服务可能包括安全服务（身份认证和权限认证），通知和命名服务，以及事务管理服务等。你可以把这些服务看做是中间件容器提供的。你可以在这个容器中部署你的组件并且这些组件可以访问和使用这些公共服务。

18.2 客户机 - 服务器计算

通过互联网访问的分布式系统通常组织成客户机 - 服务器系统。在客户机 - 服务器系统中，用户与运行在本地计算机上的程序（例如，Web 浏览器或基于电话的应用）交互。这个程序与运行在远程计算机上（例如 Web 服务器）的另一个程序交互。远程计算机提供如网页访问等服务，这些服务对于外部客户机都是可用的。如第 6 章中所述，这种客户机 - 服务器模型是一个应用程序非常普遍的体系结构模型。这种模型不受分布在许多计算机上的应用程序的限制。你也可以使用这种模型作为一种逻辑交互模型，即让客户机和服务器运行在同一台计算机上。

在客户机 - 服务器模型中，应用建模为由服务器所提供的一系列服务。客户机可以访问这些服务并提交结果给最终的用户（Orfali 和 Harkey, 1998）。客户机需要知道可用服务器的存在但不知道其他客户机的存在。客户机和服务器是独立的程序，如图 18-4 所示。该图说明了这样一种情况，有 4 台服务器（s1 ~ s4）分别提供不同的服务，每一台服务器都有一组与之关联的客户机来访问这些服务。

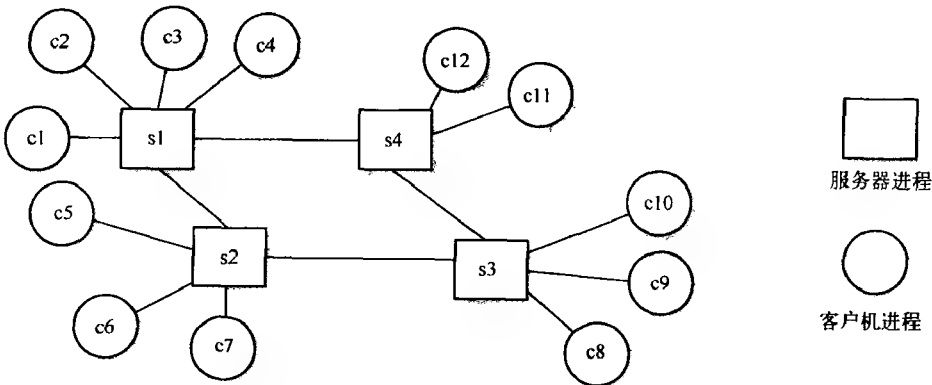


图 18-4 客户机 - 服务器交互

图 18-4 示意的是客户机和服务器进程而不是处理器。正常情况下，在单个处理器上会运行多个客户机进程。例如，在你的 PC 机上，你可能会运行邮件客户机从远程邮件服务器上下载邮

件。你也可能会运行一个 Web 浏览器与远程 Web 服务器交互，你或者运行一个打印客户端程序向远程打印机发送文件。图 18-5 说明了有 12 个逻辑的客户机（如图 18-4 所示）运行在 6 台计算机上的情况。4 个服务器进程被映射到两台物理的服务器计算机上。

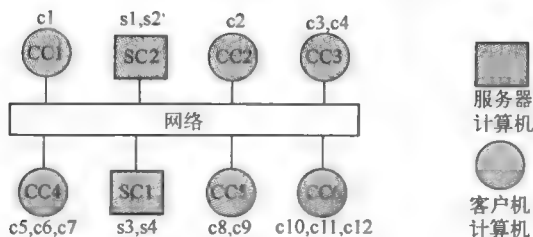


图 18-5 客户机和服务器映射到网络计算机上

许多不同的服务器进程可能会运行在同一个服务器上，但是通常服务器都是实现为多处理器系统，其中单独的服务器进程实例运行在一台机器上。负载均衡软件把客户机向服务器发送的请求分配给不同的服务器，所以每一个服务器负责相同的工作量。这就使更多的与客户机的交互事务能得以处理，而不会降低对单个客户机的响应。

客户机 - 服务器系统依赖于清晰地分离了信息表示和信息计算。信息计算是对信息的创建和处理。因此，你应该设计出分布式客户机 - 服务器系统的体系结构，让信息的表示和信息计算分布在几个逻辑的分层上，层与层之间有明确的接口。这种设计允许每一层分布在不同的计算机上。图 18-6 说明了这种模型，它将一个应用分解为 4 层：

- 表示层，表示层是关于呈现信息给用户和管理所有用户的交互。
- 数据管理层，数据管理层用来管理传给客户机和来自客户机的数据。这一层可以实现数据检查、网页生成等。
- 应用处理层，应用处理层涉及应用逻辑的实现以及向最终用户提供所需求的功能。
- 数据库层，数据库层用来储存数据和提供事务管理服务等。

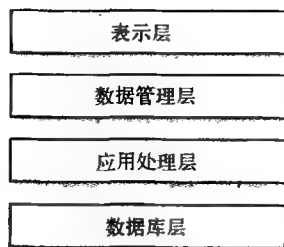


图 18-6 客户机 - 服务器应用的分层体系结构模型

接下来的章节解释不同的客户机 - 服务器体系结构如何以不同的方式分布这些逻辑的分层。客户机 - 服务器模型也强调了软件作为一种服务（Software as a Service, SaaS）的概念，这一概念成了部署软件和通过互联网访问软件的一种越来越重要的方法。18.4 节将会讨论这一问题。

18.3 分布式系统的体系结构模式

如本章的引言中所述，分布式系统的设计者必须要组织他们的设计以便在系统的性能、可依赖性、信息安全性和可管理性之间找到平衡。因为没有能适用于所有环境的一种通用的系统组织模型，所以出现了不同的分布式体系结构风格。当设计一个分布式应用时，你应该选择一种能支持你的系统中关键的非功能性需求的体系结构风格。

在这一节，讨论了 5 种体系结构风格：

1. 主从体系结构，这种体系结构常用在实时系统中，在实时系统中需要保证交互的反应时间。
2. 两层客户机 - 服务器体系结构，这种体系结构常用于简单的客户机 - 服务器系统，也用于由于信息安全原因集中化系统是至关重要的情况下。在这种情况下，客户机和服务器之间的

通信通常是经过加密的。

3. 多层客户机 - 服务器体系结构, 这种体系结构用于服务器要处理大量的事务的时候。
4. 分布式组件体系结构, 这种体系结构用于不同系统和数据库的资源需要合并的时候。或者作为多层客户机 - 服务器系统实现模型。
5. 对等体系结构, 这种体系结构用于当客户机交换本地存储信息而服务器作为在客户机之间相互介绍认识的时候。也用于要必须进行大量的独立计算的时候。

18.3.1 主从体系结构

分布式系统的主从体系结构通常用于实时系统, 在实时系统中有独立的处理器分别针对从系统的环境中获取数据、数据处理、数据计算以及执行器管理。就像第 20 章将讨论的, 执行器是由软件系统所控制的设备, 它们用来改变系统的环境。例如, 一个执行器可能控制一个阀门并把阀门的状态由“开”变为“关”。主进程通常负责计算、协调和通信, 并控制从进程。从进程用于执行专门的行为, 比如从传感器阵列中获取数据。

图 18-7 说明了这种体系结构模型。这是一个城市交通管理系统的模型, 这个模型有三个运行在不同处理器上的逻辑进程。主进程是控制室进程, 它与负责收集交通数据和管理交通信号灯的独立的从进程通信。

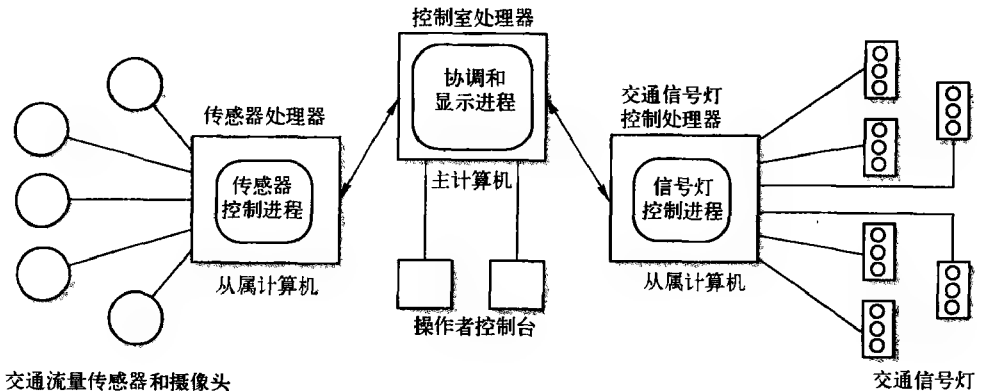


图 18-7 交通管理系统的主从体系结构

一组分布式传感器收集交通流量信息。传感器控制进程周期性地轮询传感器来获取交通流量信息并整理这些信息以便进一步处理。传感器处理器本身也被主进程周期性地轮询来提供信息。主进程负责向操作者显示交通状况, 计算交通信号灯序列, 接受操作者的指令来修改这些序列。控制室系统发送指令给交通信号灯控制进程, 交通信号灯进程把这些命令转化为信号来控制交通信号灯硬件。主控制室系统本身组织为一个客户机 - 服务器系统, 客户机进程运行在操作者的控制台程序。

我们可以在这样的情形中使用分布式系统的主从模型, 即能够预测所需要的分布式处理和计算可以很容易地定位到从处理器上。这种情况一般发生在实时系统中, 在实时系统中最重要的是要满足处理的时限。可以使用从处理器于计算密集型操作, 比如信号处理和系统控制的设备管理。

18.3.2 两层客户机 - 服务器结构

在 18.2 节中讨论了客户机 - 服务器系统的一般形式, 在客户机 - 服务器系统中一部分应用

系统运行在用户的计算机上（客户机），另一部分运行在远程计算机上（服务器）。此外，还提出了一种分层应用模型（见图 18-6），其中系统中的不同层可能会在不同的计算机上执行。

两层客户机-服务器体系结构是客户机-服务器体系结构最简单的形式。系统实现为一个独立的逻辑服务器加上不确定数目的使用服务器的客户机。图 18-8 显示了这种体系结构模型的两两种形式：

1. 瘦客户机模型，其表示层在客户机端实现，其他层（数据管理、应用处理和数据库）在服务器上实现。客户机软件可能是专门在客户机上编写的程序以处理表示。不过，更常见的情况是在客户机上使用 Web 浏览器来表示数据。

2. 胖客户机模型，一部分或者所有的应用处理都在客户机上执行。数据管理和数据库功能在服务器上实现。

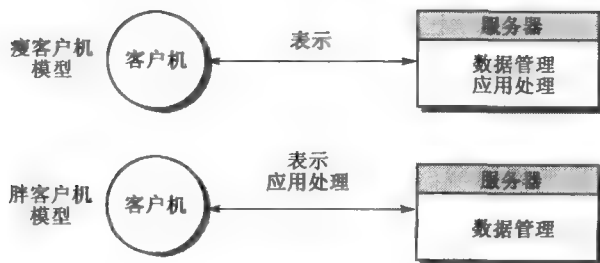


图 18-8 瘦客户机和胖客户机的体系结构模型

瘦客户机模型的优点是管理客户机很简单。如果存在很多的客户机那么管理就是最主要的问题，因为在所有的客户机上安装新的软件是非常困难和昂贵的。如果使用 Web 浏览器作为客户端就不需要安装任何软件。

瘦客户机模型的缺点是它将繁重的处理负荷都放在了服务器和网络上。服务器负责所有的计算，这将导致客户机和服务器之间的网络上很大的流量。使用这种模型来实现一个系统可能需要在网络和服务器容量上投入更多。不过，浏览器可以通过在网页中执行脚本语言（比如 JavaScript）来执行一些本地的处理。

胖客户机模型利用运行客户机软件的计算机上的处理能力，并把部分或所有的应用处理和表示分布到客户机上。实际上，服务器就是一个事务服务器，即管理所有的数据库事务。数据管理是简单的因为不需要管理客户机和应用处理系统之间的交互。当然，胖客户机模型的问题是它需要额外的系统管理来部署和维护客户机上的软件。

使用胖客户机体系结构的一个例子是银行的 ATM 系统，ATM 系统向用户提供现金和其他的银行服务。这里 ATM 是一个客户机，服务器是一个典型的大型主机，在它上面运行客户账户数据库。大型主机是一个为事务处理而设计的强大的机器。因此这个大型主机可以处理由 ATM 机、其他的柜台系统和在线银行产生的大量事务。自动取款机上的软件执行很多与用户相关的处理，这些处理与交易有关。

图 18-9 显示了 ATM 系统组成的一个简化版本。注意，ATM 不是直接连接到客户账户数据库上，而是连到一个远程处理（TP）监控器上。远程处理监控器是一个中间件系统，它组织远程客户机和数据库间的通信并序列化客户机交易以便数据库处理。这确保了交易的独立并且不会干涉其他交易。使用事务序列化的好处是系统能从错误中恢复而不至于造成系统数据崩溃。

一方面，胖客户机模型比瘦客户机模型在分布处理上更有效；另一方面，它也使系统管理更复杂。应用功能分散在许多不同的计算机上。当应用软件不得不变更时，就需要对每个

客户机计算机进行重新安装。当系统中有数以百计的客户机时，就可能造成比较大的成本开销。该系统必须要设计得能支持远程软件更新，关闭所有的系统服务直到客户软件更新完毕是很有必要的。

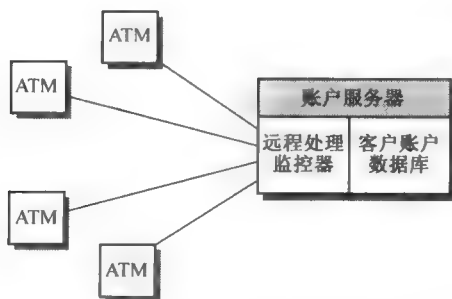


图 18-9 ATM 系统的胖客户机体系结构

18.3.3 多层客户机 - 服务器结构

两层客户机服务器方法的根本问题是系统中的逻辑层（表示层，应用处理层，数据管理层和数据库层）必须要映射到两个计算机系统上：客户机和服务器。如果选择的是瘦客户机模型，则可能导致伸缩性和性能的问题，若选择的是胖客户机模型，则可能有系统管理上的问题。为了避免其中的一些问题，可以使用一种“多层客户机 - 服务器”的体系结构。在这种体系结构中，系统中的不同层（表示层，数据管理层，应用处理层和数据库层）是在不同处理器上执行的独立的进程。

互联网银行系统（见图 18-10）是使用多层客户机 - 服务器体系结构的例子，这个系统中有 3 层。银行的客户数据库（通常在如上所述的一个大型计算机上）提供数据库服务，一个 Web 服务器提供（如网页的生成）数据管理服务和一些应用服务。诸如现金转账、生成银行结算单和工资单等应用服务在 Web 服务器上实现，并作为脚本在客户机上执行。用户计算机和浏览器构成客户机。这种系统是可扩展的，因为当客户数量增加的时候增加服务器（增加扩展）相对还是比较容易的。

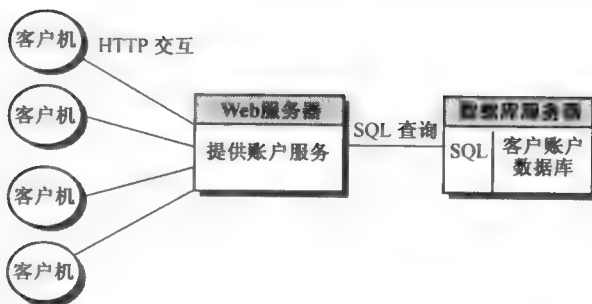


图 18-10 网上银行系统的三层体系结构

在这种情况下，使用三层体系结构可以对 Web 服务器和数据库服务器之间信息传递进行优化。这两个系统之间的通信无需基于互联网标准，可以使用比较低层的数据交换协议。使用支持 SQL（结构化查询语言）数据库查询的高效中间件从数据库取回信息。

在系统中添加更多的服务器，可以把三层的客户机 - 服务器模型扩展成多层结构。这可能

包括使用 Web 服务器管理数据,使用独立的服务器处理应用程序以及数据库服务。多层系统适合于当应用程序需要到不同的数据库中存取数据的情况。在这种情况下,或许需要在系统中增加一个集成的服务器。这个集成服务器收集分布式数据并将其传递给应用程序,就好像数据是在一个单一的数据库中一样。如下一节中将要提到的,分布组件体系结构可以用来实现多层客户及服务器系统。

在多个服务器上分布应用处理的多层客户机-服务器系统本质上比两层的体系结构有更强伸缩性。应用处理是系统的最易变的部分,由于它位于系统的中间,它也可以很容易修改。在某些情况下,处理可以分布到不同的应用逻辑和数据管理服务器中,以便谋求更快的客户响应。

当选择最适当的分布式体系结构时,客户机-服务器体系结构的设计者必须考虑到多种因素。客户机-服务器体系结构的适应条件如图 18-11 所示。

体系结构	应用
两层瘦客户机 C/S 体系结构	遗留系统应用,当将应用处理过程和数据管理分离是不切合实际的时候使用。客户机把它们当做服务去访问,如在 18.4 节中所讨论的那样 计算密集的应用,例如编译器,很少或根本没有数据管理 数据密集的应用(浏览和查询),很少或根本没有应用处理。浏览 Web 页面是最为普通的使用此体系结构情形的例子
两层胖客户机 C/S 体系结构	其应用处理是在客户机上由 COTS(如微软的 Excel)所提供的这类应用 需要密集的数据处理(如数据可视化)的应用 移动应用,互联网链接无法保证的应用。某些使用来自数据库的缓存信息的局部处理因而是可能的
三层或多层 C/S 体系结构	大规模的应用,具有成百上千个客户机的应用 数据和应用都是易变的一类应用 数据是由多处经过集成而得的一类应用

图 18-11 客户机-服务器体系结构模式的应用

18.3.4 分布式组件体系结构

如图 18-6 所示,通过把进程组织到各个层中去,系统的每一层可以实现为一个独立的逻辑服务器。这个模型对许多类型的应用来说能工作得很好。然而,它对系统设计者的灵活性是一个很大的限制,因为设计者必须决定每一层应该包括哪些服务。然而在实际中,一个服务到底是数据管理服务、应用服务还是数据库服务不总是很明确的。设计者必须还要规划系统的伸缩性,当有较多的客户机增加到系统中时,他们要提供服务器能被复制的一些方法。

分布式系统设计的更通用方法是把系统设计为一系列服务,不需要试图把这些服务定位到系统中的某一层上。每一个服务或者相关的一组服务是使用独立的组件实现的。在分布组件体系结构中(见图 18-12),系统被组织成一系列交互的组件或对象。这些组件给出了它们所提供的一个接口。其他组件通过中间件使用远程过程或方法调用请求这些服务。

分布式组件系统依赖于中间件来管理组件间的交互,弥合组件间传递的不同类型参数之间的差异,并提供一系列公共服务供应用组件使用。CORMA(Orfali 等,1997)是这种中间件早期的一个例子,但是现在应用的不是很广泛了。它已在很大程度上被专有的软件所取代,比如企业 Java Beans(EJB)或者.NET。

使用分布组件模型来实现分布式系统的好处如下:

1. 它允许系统设计者延迟决断在哪里和如何提供服务。提供服务的对象可以在网络任何节点上运行。没有必要事先决定一个服务是数据管理层的一部分还是应用层的一部分等。

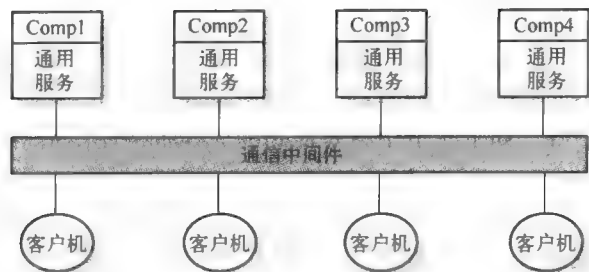


图 18-12 分布式组件体系结构

- 2. 它是一个非常开放的体系结构，允许新的资源根据需要增加进来。可以很容易地添加新的系统服务而不会对现有的系统有很大的影响。
- 3. 系统具有很好的柔性和可扩展性。当系统负荷增加时，增加新的组件或者增加可复用的组件，这样做不至于引起系统其他对象的混乱。
- 4. 通过组件在网络上的迁移达到对系统的动态配置是有可能的。这在服务需求模式不确定的场合中可能是很重要的。一个提供服务的组件能迁移到请求服务的组件所在的同一台处理器上，这样可以改善系统的性能。

分布式组件体系结构可以作为一个逻辑模型来构造和组织系统。在这种情况下，要考虑该如何提供应用功能，把这些功能按照服务或服务组合的形式给出。然后设计如何利用分布式组件来提供这些服务。举例来说，在一个零售应用中可能有关于库存控制、客户通信、货物订购等应用组件。

数据挖掘系统是这种系统的一个很好的例子，在数据挖掘系统中分布式组件体系结构是最合适使用的体系结构。数据挖掘系统寻找在多个不同数据库中数据间的关系（见图 18-13）。数据挖掘系统通常从许多独立的数据库中得出信息，执行计算密集型处理以及图形化的显示结果。

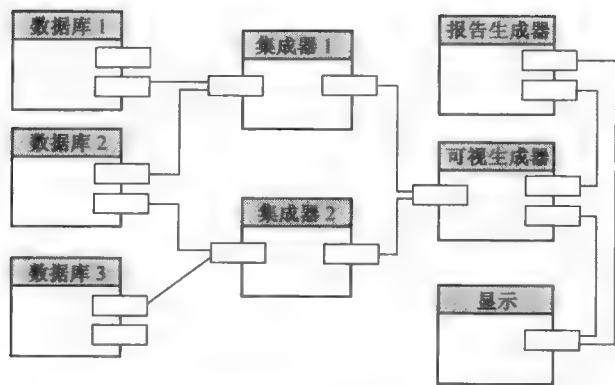


图 18-13 数据挖掘系统的分布式组件体系结构

这种数据挖掘应用的一个例子可能是一个出售食物和书籍的零售业系统。市场部门想要寻找客户购买食物和书籍之间的联系。例如，相当多的人在买比萨的同时也会购买犯罪小说。通过这种知识，企业可以在新小说出版的时候明确地向购买具体食物的客户提供新小说的信息。

在这个例子中，每个数据库可以封装成一个分布式组件，提供的接口提供只读访问功能。集成器组件专注于一种特别类型的关系，它们从所有的数据库中收集信息来试着推导出这种关系。可能有一个集成器关注食品销售的季节性变化关系，还有一个关注不同类型商品之间的销售关系。

可视化组件与集成器组件交互来给出所发现的关系的可视化效果或报告。由于所要处理的数据量巨大, 可视化组件通常图形化地表示它们的结果。最后, 显示组件可能负责向客户端传递图形化模型以便最终的表现。

对这一类型应用, 分布式组件体系结构比客户机-服务器体系结构更适合, 因为你可以向系统中增加新的数据库而不会有很大的影响。每一个新的数据库只是加入另一个访问分布式组件。数据库访问组件提供一个简化的界面以控制数据的操作。访问的数据库可以存在于不同的机器上。该体系结构也使得通过添加新的集成组件来挖掘新的关系类型变得简单。

分布式组件体系结构有两个主要缺点:

1. 其设计比客户机-服务器系统更复杂。多层客户机-服务器系统看起来似乎更直观。它反映了很多人的事务, 人们从其他专门提供这些服务的人那里请求和接收服务。相反, 分布式组件体系结构对于人们来说很难想象和理解。

2. 分布组件系统的标准化的中间件还从来没有被软件社会所接受。不同的提供商比如微软和 Sun 公司开发出了不同的也是相互不兼容的中间件。这种中间件非常复杂, 依赖它们会增加整个分布组件系统的复杂度。

因为这些问题, 在很多情况下, 面向服务的体系结构 (在第 19 章讨论) 取代了分布组件系统体系结构。不过, 分布组件系统要比面向服务的系统性能更优越。RPC 通信通常要比面向服务系统使用的基于消息的交互更快。因此基于组件的体系结构更适合高通量的系统, 这种系统中大量的事务需要及时处理。

18.3.5 对等体系结构

本章的前几节提到的客户机-服务器计算模型对于提供服务的服务器和接收服务的客户机有明确的区分。这种模型通常导致系统不均衡的负载分布, 其中服务器要比客户机做更多的工作。这就会导致花费很多的组织工作在服务器的性能上, 然而在成千上万的用来访问系统服务器的 PC 机上却有很多没有被使用的处理能力。

对等系统 (p2p) 是分散式系统, 它的计算是由网络上的任一个节点来承担的, 至少从理论上讲, 是不存在客户机和服务器之间的区别的。对于对等式应用, 总的系统设计建立在潜在的巨大的计算机网络的计算能力和存储资源的优势上。保证节点间能有效通信的标准和协议都是嵌入于应用系统本身的, 每个节点必须运行一个这样的应用的拷贝。

对等技术几乎都是用于个人系统 (Oram, 2001)。例如, 我们使用基于 Gnutella 和 BitTorrent 协议的文件交换系统在个人计算机上共享文件, 又例如, 我们使用即时消息系统比如 ICQ 和 Jabber 在两个用户之间通信而无需中间服务器提供支持。SETI@home 是一个长期项目, 它在家庭个人计算机上处理来自射电望远镜传来的数据, 搜索宇宙中生命的迹象。Freenet 是一个分散式数据库, 设计用于匿名的信息发布, 避免权力机构对这些信息的查禁。利用 IP 的声音电话服务 (VOIP), 比如 Skype, 依赖于电话呼叫或会议的相关各方之间的对等通信。

然而, 企业也利用对等系统来更好地使用它们个人计算网络的能力 (McDougall, 2000)。Intel 公司和 Boeing 公司都实现了 p2p 系统用于强计算应用。这样就利用了本地计算机没有使用的处理能力。工程计算可以整夜地运行在没有使用的桌面电脑上, 这取代了购买昂贵的高性能硬件。企业也在广泛地使用商务 p2p 系统, 比如短消息系统和 VOIP 系统。

在两种环境下系统适合使用对等体系结构模型:

1. 系统是计算密集的并且有可能把需求的处理分成许多独立的计算。例如, 支持药物探索计算的对等系统分散了通过分析大量的分子来寻找治疗癌症潜在方法的计算, 以此来观察这些分子是否具有所需求的抑制癌症增长的特性。每一个分子都可以独立地分析, 所以系统中的同

行不需要通信。

2. 系统主要涉及个人计算机在网络上的信息交换，没有必要将这些信息集中管理和储存。这种应用的例子包括允许对等体交换比如音乐和视频等本地文件的文件共享系统和支持计算机间语音和视频通信的电话系统。

原则上，在对等系统中网络上的每一个节点都能够看到每个其他节点，可以与之建立连接，并可以与之直接交换数据。当然，实际过程中这是不可能的，所以节点都是以区域来组织的，通过某些节点作为桥梁来与其他区域中的节点群建立连接。图 18-14 示意了这种分散式 p2p 体系结构。

在分散式体系结构中，网络中的节点不仅仅是一个功能单元，它还是通信的转换器，能够从一个节点到另一个节点路由数据和控制信号。例如，假设图 18-14 代表一个分散式文件管理系统。该系统由一个研究人员团体来共享文件，每一个该团体中的成员维护他自己的文件库。然而，当一个文件被检索到，这个检索文件的节点还要使之对所有节点都是可用的。

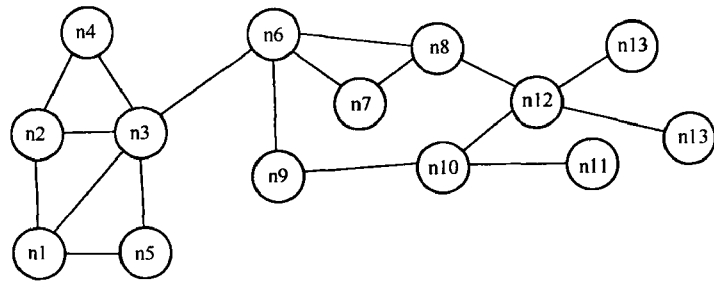


图 18-14 集中式 p2p 体系结构

如果有谁需要一个储存在网络中某个地方的一个文件，就发出一个搜索命令给本区域中的节点。这些节点检查是否它们有所要的文件，如果有，则将信息发回给请求者，如果它们没有，它们就将此搜索路由到其他节点。因此，如果 n1 发出对某个存储于节点 n10 上的文件的搜索命令，该搜索路由经过节点 n3、n6 和 n9，到达 n10。当文件最终得以发现，节点先持有这个文件接着通过对等连接把文件直接发送给请求节点。

这种分散式体系结构的优点在于它是高度冗余的，所以它是容错的而且是容许节点从网络中断开。然而，系统的缺点是相同的搜索会被很多不同的节点来处理，这种重复的对等通信是开支很大的。

一种替代的 p2p 体系结构模型是不同于单纯 p2p 体系结构的半分散式体系结构，即在网络中，有一个或多个节点充当服务器来提供对其他节点的通信。这减少了节点之间的流量，图 18-15 说明了这个模型。

在半集中式体系结构中，服务器（有时叫做超级对等体）的作用是帮助建立网络中两个节点之间的联系，或者是协调计算的结果。例如，如果图 18-15 代表一个即时消息系统，那么网络节点与服务器通信（用虚线表示）来寻找哪些其他节点是可用的。一旦发现了它们，就能够建立它们之间的直接通信而与服务器的连接就不再需要了。因而，节点 n2、n3、n5 和 n6 处于直接通信状态。

在 p2p 计算系统中，把处理器密集的计算分布到很多个节点上去，通常是要有一些节点作为超级对等体，它们的作用是将工作分布到各个其他节点上并收集和检查计算的结果。

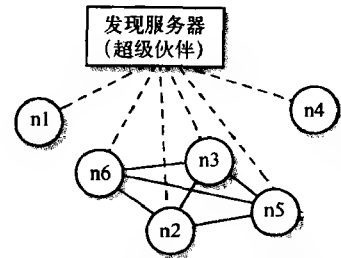


图 18-15 半集中式 p2p 体系结构

对等体系结构允许网络能力的有效使用。不过，阻碍这种体系结构应用的最主要的问题是信息安全和信任问题。对等通信需要开放你的电脑直接与其他对等体交互，这意味着这些系统有潜在可能访问你的任何资源。为了应对这种情况，需要组织你的系统来保护这些资源。如果没有做好这些，你的系统就是不安全的。

问题也可能发生在网络上的对等体故意地做违法的事。例如，曾有这样的情况，认为自己的版权遭到滥用的音乐公司其实故意提供“有毒的对等体”。当其他的对等体下载他们认为的音乐文件时，真正交付的文件却是流氓软件，这种流氓软件可能是版本故意损坏的音乐或者是对用户侵犯版权的一个警告。

18.4 软件作为服务

在前几节，讨论了客户机-服务器模型以及功能是如何在客户机和服务器之间分布的。为了实现一个客户机-服务器系统，可能需要在客户机上安装程序并管理用户界面。例如，一个像 Outlook 或 Mac Mail 的邮件客户端，在自己的计算机上提供邮件管理的特征。这避免了所有的进程都运行在服务器上的瘦客户机系统的一些问题。

不过，通过使用当前的浏览器作为客户端软件的方法，服务器超负荷的问题得到了明显的改善。Web 技术，比如 AJAX (Holdener, 2008)，支持网页表示的有效管理和通过脚本语言的本地计算。这意味着浏览器可以作为客户端配置和使用，有很强的本地处理能力。应用软件可以被看做远程服务，可以被任何运行标准浏览器的设备访问。众所周知的一个例子是基于 Web 的邮件系统，例如 Yahoo! 和 Gmail 以及办公应用比如 Google docs。

SaaS 这个概念涉及远程地托管软件以及通过互联网提供对软件的访问。SaaS 的关键要素如下：

1. 软件部署在一台服务器上（更一般的情形是很多台服务器）并且可以通过 Web 浏览器访问。软件不是部署在本地 PC 机上的。
2. 软件提供商拥有和管理该软件，而不是使用软件的机构拥有它。
3. 用户为使用该软件付费，根据他们的使用量或者包年包月来支付。有时，该软件免费地给所有人使用，但是用户必须同意接受广告，因为广告为软件服务提供资金。

对于软件使用者，SaaS 的好处在于软件管理的花费转移给了提供商。软件提供商负责修复错误和升级软件，处理操作系统平台的变化，确保硬件性能满足需求。软件许可管理成本是零。如果某个人有多台计算机，不需要为所有的计算机注册软件。如果一个软件应用只是偶尔才用到，那么按每次使用支付要比购买这个应用更便宜。比如手机这种移动设备可以在世界的各个地方访问软件。

当然，这种软件提供模型也有一些缺点。主要问题可能是数据传输到远程服务上的花费。数据传输在网络传输速度慢时，大流量数据要花费大量的时间。你或许还要根据传输量向服务提供商支付费用。其他问题包括缺少对软件演化的控制（提供商可能在他们觉得合适的时候来更新软件），以及法律法规的问题。许多国家有专门的法律来规定数据的存储、管理、维护和访问，移动数据到远程的服务可能会触犯这些法律。

SaaS 的概念和第 19 章要讨论的面向服务的体系结构 (SOA) 显然是很相关的但又不一样：

1. SaaS 是在远程服务器上提供功能客户机通过 Web 浏览器访问的一种方法。服务器在交互会话期间维持用户的数据和状态。事务常常是长事务（例如，编辑文件）。
2. SOA 是把软件系统构建为一系列单独的无状态服务的方法。这些服务或许由多个提供商提供并且可能是分布的。典型地，事务是短事务，服务被调用，做一些处理，接着返回结果。

SaaS 是向用户传送应用功能的方法，而 SOA 是应用系统的一种实现技术。使用 SOA 实现

的功能不需要向用户显示为服务。同样的，用户服务也不需要使用 SOA 来实现。不过，如果 SaaS 是使用 SOA 实现的，那么应用程序就可能使用服务 API 来访问其他应用程序的功能。它们然后可以整合到一个更复杂的系统。这叫做混合，代表了软件复用和快速软件开发的另一种方法。

从软件开发的角度来看，服务的开发过程与其他的软件开发有很多的相似之处。不过，服务的构建通常不是由客户的需求驱动的，但是服务提供商假设用户会需要什么。因此软件需要能够在得到用户的需求反馈后很快地演变。因此，敏捷开发和增量式交付是软件作为服务开发常用的一种方法。

当要实现 SaaS 的时候必须要考虑可能会有许多不同机构的软件用户。必须要考虑到以下 3 点：

1. **可配置性** 如何为每一个机构的集体需求配置软件？
2. **多重租赁性** 如何让每一个用户感觉到他们正在使用他们自己的系统副本，而同时能高效地使用系统资源？
3. **可扩展性** 如何设计系统以至于可以扩展来容纳不可预测的大量用户。

在 16 章讨论的产品线体系结构的概念是配置系统满足用户的多重但不完全相同的需求的一种方法。从一种通用的系统开始，并根据每个用户的具体需求调整系统。

不过，这对于 SaaS 是不起作用的，因为这将意味着为使用这种软件的每个机构开发服务的一个不同副本。所以，需要在系统中设计可配置性并提供一个配置接口以便允许用户指定他们的性能。当软件使用的时候使用这些来动态的适应软件的行为。配置设施可能会允许以下内容：

1. 品牌化，为每一个机构的用户提供一个反映他们机构的接口。
2. 业务规则和工作流，每个机构定义自己的服务和数据使用的规则。
3. 数据库扩展，每个机构定义如何将通用服务数据模型扩展为能满足特殊的需求的模型。
4. 访问控制，服务的客户为他们的员工建立个人账户并为他们的每个用户定制可访问的资源 and 功能。

图 18-16 说明了这种情况，这个图显示了应用服务的 5 个用户，他们分属于这个服务供应商的 3 个不同的客户。用户与服务的交互是通过客户资料文档中为员工所定义的服务配置进行的。

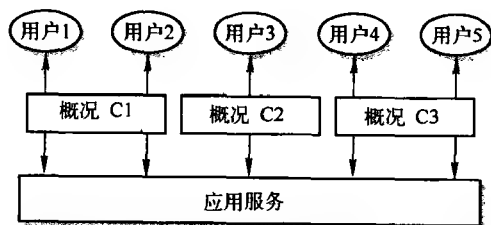


图 18-16 软件系统作为服务的配置

多重租赁性是这样一种情况，许多不同的用户访问相同的系统并且定义的系统体系结构允许系统资源的有效共享。不过，对于每个用户来说必须感觉他们在唯一地使用系统资源。多重租赁性涉及系统设计所以在系统功能和系统数据之间要有一个绝对的分离。因此，应该把系统设计成所有的操作都是无状态的。数据要么应该由客户机提供，要么应该在存储系统或数据库中可以被所有的系统实例访问。关系数据库对于提供多重租赁性是不合适的，很多大型服务提供商比如 Google 已经为用户数据实现了一个更简单的数据库。

多重租赁的系统的一个特别的问题是数据管理。提供数据管理最简单的办法是让每一个客

户有自己的数据库，这个数据库可以按照他们的想法使用和配置。不过，这需要服务提供商维护许多不同的数据库实例（每个客户一个）并使这些数据库在需求的时候是可用的。这对于系统性能来说是非常低效的，并且增加了系统的整体成本。

另外一种方法是，服务提供商可以使用一个单一的数据库，在数据库中虚拟地隔离不同的用户。图 18-17 说明了这一点，可以看到数据库入口还有一个“租客标识符”，这个标识符将数据入口关联到特别的用户上。通过使用数据库视图，你可以为每一个服务的客户提取出数据入口，以此提供给此客户单位中的用户一个虚拟的个人数据库。使用上面提到的配置特征，可以扩展这种方法来满足具体的客户需要。

租户	关键字	名字	住址
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

图 18-17 多重租赁数据库

可扩展性是系统可以处理不断增加的用户数量而不降低向每个用户提供的整体的 QoS（服务质量）的一种能力。通常，当在 SaaS 的环境下考虑可扩展型时，我们采用的是“增加扩展”而不是“增强扩展”。回忆一下，“增加扩展”是指增加更多的服务器，由此也增加可以并行处理的事务数量。可扩展性是个复杂的话题，这里不展开详细讨论，但是可扩展软件的一般准则如下：

1. 所开发的应用要使得每一个组件都实现为简单的无状态服务，每一个服务都可以运行在任何一个服务器上。因而在单一事务过程中，用户可以与运行在多个不同服务器上的相同服务的多个实例交互。
2. 使用异步交互来设计系统，这样应用程序就不需要等待交互的结果（比如一个读请求）。这将允许应用程序在等待交互的结束的同时可以继续执行现在的工作。
3. 管理资源，比如网络和数据库连接，将所有资源作为一个公用池，因此不让任何一个服务器游离在资源之外。
4. 设计你的数据库以允许细粒度的上锁。即当只有记录的一部分在使用时没必要锁定数据库中的所有记录。

SaaS 的概念是分布式计算的一个主要模式转变。与在自己的服务器上管理多个应用程序的组织方式相比，SaaS 允许不同的厂商从外部提供这些应用程序。我们正在经历从一个模型到另一个模型的转换，未来，它可能在企业软件系统工程方面产生巨大的影响。

要点

- 分布式系统的好处是在于它可以调整以应对日益增长的需求，可以持续地向用户提供服务（即使系统的某些部分失败），而且使资源得到共享。
- 分布式系统的设计需要考虑的问题包括：透明性，开放性，可扩展性，信息安全性，服务质量和失败管理。
- 客户机-服务器系统是分布式系统，其中系统构建为层次结构，表现层在客户机上实现。服务器提供数据管理、应用和数据库服务。
- 客户机-服务器系统可能有许多层，系统的不同层分布在不同的计算机上。

- 分布式系统的体系结构模式包括主从体系结构，两层和多层客户机 - 服务器结构，分布式组件体系结构和对等体系结构。
- 分布式组件体系结构需要中间件来处理组件间的通信并允许在系统中增加和删除组件。
- 对等体系结构是分散式体系结构，其中没有区分客户机和服务器。计算可以分布在不同机构的很多系统中。
- 软件作为服务是把应用部署为瘦客户机 - 服务器系统的一种方法，其中客户端是 Web 浏览器。

进一步阅读材料

《Middleware: A model for distributed systems service》这是一篇非常优秀的综述性文章，它总结了中间件在分布式系统中的作用，并讨论了中间件能提供的服务范围 (P. A. Bernstein, Comm. ACM, 39 (2), February 1996)。http://dx.doi.org/10.1145/230798.230809。

《Peer-to-Peer: Harnessing the Power of Disruptive Technologies》尽管这本书并没有太多关于 p2p 体系结构的内容，但是它是关于 p2p 计算的一个很棒的介绍，也讨论了在多个 p2p 系统中所采用的结构和方法 (A. Oram (ed), O' Reilly and Associates, Inc. 2001)。

《Turing software into a service》这是一篇综述文章，讨论了面向服务计算的一些准则。不像很多此主题的其他文章，它揭示了隐藏于很多标准背后的这些准则 (M. Turner, D. Budgen and P. Brereton, IEEE Computer, 36 (10), October 2003)。http://dx.doi.org/10.1109/MC.2003.1236470。

《Distributed Systems: Principles and Paradigms, 2nd edition》是一本讨论分布式系统设计和实现的综合性教科书。不过，书中并没有包括面向服务范式的讨论 (A. S. Tanenbaum and M. Van Steen, Addison-Wesley, 2007)。

《Software as a Service; The Spark that will Change Software Engineering》讨论 SaaS 的到来将会推动所有软件开发使用迭代模型的一篇短文 (G. Goth, Distributed System Online, 9 (7), July 2008)。http://dx.doi.org/10.1109/MDSO.2008.21。

练习

- 18.1 如何理解“可扩展性”？讨论“增强扩展”和“增加扩展”的区别并解释何时使用这些不同的可扩展性方法？
- 18.2 解释为什么分布式软件系统要比所有的系统功能都实现在单一计算机上的集中式软件系统更复杂？
- 18.3 用一个远程过程调用的例子来说明中间件是如何在分布式系统中协调各计算机之间的交互的？
- 18.4 在客户机 - 服务器系统体系结构中，胖客户机和瘦客户机方法的根本差别在哪里？
- 18.5 假如要求你设计一个需要强身份认证和权限认证的信息安全系统。该系统必须保证系统中各个部分之间的通信不能被入侵者拦截和读取。提出该系统最合适的客户机 - 服务器体系结构，并解释为什么你如此设计，在客户机和服务器之间功能应该如何分布？
- 18.6 假设要开发一个股票信息系统，向客户提供对公司信息的访问并能够利用仿真系统对各种投资情形做出评估。不同的客户会根据他们的经验而采取不同的投资方式，而且购买的股票类型也是不同的。为系统提出一个客户机 - 服务器体系结构，指出各个功能是在哪里实现的，并对该模型做出一些判断。
- 18.7 使用分布式组件的方法，为国家剧院订票系统设计一个体系结构。用户可以查询空座以及预订一组剧场的座位。系统应当支持退票，因此，人们可以退掉他们的票以便剧院在最后一分钟将票转售给其他客户。
- 18.8 分别给出分散的体系结构和半集中的对等体系结构的两个优点和两个缺点。
- 18.9 解释为什么以将软件部署为一种服务可以减少公司的 IT 支持花销。

- 18.10 你的公司想要从使用桌面应用程序转变为以服务的方式访问远程相同的功能。确定 3 个可能出现的风险并建议如何可以减少这些风险。

参考书目

- Bernstein, P. A. (1996). 'Middleware: A Model for Distributed System Services'. *Comm. ACM*, **39** (2), 86–97.
- Coulouris, G., Dollimore, J. and Kindberg, T. (2005). *Distributed Systems: Concepts and Design, 4th edition*. Harlow, UK.: Addison-Wesley.
- Holdener, A. T. (2008). *Ajax: The Definitive Guide*. Sebastopol, Calif.: O'Reilly and Associates.
- McDougall, P. (2000). 'The Power of Peer-To-Peer'. *Information Week* (August 28th, 2000).
- Neuman, B. C. (1994). 'Scale in Distributed Systems'. In *Readings in Distributed Computing Systems*. Casavant, T. and Singal, M. (ed.). Los Alamitos, Calif.: IEEE Computer Society Press.
- Oram, A. (2001). 'Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology'.
- Orfali, R. and Harkey, D. (1998). *Client/server Programming with Java and CORBA*. New York: John Wiley & Sons.
- Orfali, R., Harkey, D. and Edwards, J. (1997). *Instant CORBA*. Chichester, UK: John Wiley & Sons.
- Pope, A. (1997). *The CORBA Reference Guide: Understanding the Common Request Broker Architecture*. Boston: Addison-Wesley.
- Tanenbaum, A. S. and Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms, 2nd edition*. Upper Saddle River, NJ: Prentice Hall.

面向服务的体系结构

目标

本章的目标是介绍面向服务的软件体系结构，这是一种使用 Web 服务来构建分布式应用程序的方式。读完本章，你将了解以下内容：

- Web 服务的基本概念、Web 服务标准，以及面向服务的体系结构；
- 介绍旨在产生可复用的 Web 服务的软件工程过程；
- 介绍服务组成的概念，这是一种作为面向服务的应用开发的手段；
- 理解为什么业务过程模型可作为一个面向服务系统的设计基础。

在 20 世纪 90 年代，网络技术的发展彻底改变了机构的信息交流方式。客户计算机可以访问它们机构外的远程服务器来获得信息。但是，这种访问完全是通过 Web 浏览器进行的，要想使用其他程序来对信息库进行直接访问是不实际的。也就是说，在服务器之间进行随意的连接（比如一个程序从不同的提供商查询多个目录）是不可能做到的。

为了解决这个问题，人们提出了 Web 服务的概念。使用 Web 服务，机构通过定义和建立一个 Web 服务界面就可让自己的信息被别的程序访问。这个界面定义可用的数据和如何访问这些数据。更一般的情况，Web 服务是一个标准的计算资源或信息资源的表示，这些资源可以被其他程序使用。这些可能是信息资源，例如一个零件目录；也可以是计算机资源，例如一个专门的处理器；或者是存储资源，例如，存档服务能够实现对商店数据的长期、可靠存储，依据法律这些组织数据必须保持多年。

Web 服务是更一般的服务概念的一个实例，对于一般的服务概念，由 Lovelock 给出的定义（Lovelock 等，1996）是这样的：

由一个团体向另一个团体提供的行动或能力。尽管这个过程可能是与一个有形的产品联系在一起，但是能力本质上是无形的，一般不会产生对任何作品因素的拥有权。

因而，服务的本质是服务的提供独立于使用服务的应用（Turner 等，2003），服务提供者能开发专业性服务并提供面向不同机构中的某个范围内的服务用户。

面向服务的体系结构（SOA）是一种开发分布式系统的方法，分布式系统的系统组件是单机服务，这些服务在不同地理位置的计算机上执行。基于 XML 标准的协议，例如 SOAP 和 WSDL，这些设计支持服务通信和信息交换。因此，服务是与平台和实现语言无关的。软件系统可以通过使用本地的服务和不同提供商提供的外部服务来构建，这些服务之间可以做到无缝交互。

图 19-1 封装了 SOA 的思想。服务的提供商设计 and 实现服务并定义了这些服务的接口。他们也发布这些服务的有关信息到能访问到的注册表。那些希望利用某项服务的服务请求者（有时叫做服务客户）发现某个服务的描述，从而也定位服务的提供者。然后他们能够将自己的应用绑定到特定的服务并使用标准的服务协议与之通信。

从一开始，伴随着技术发展，一直有一个活跃

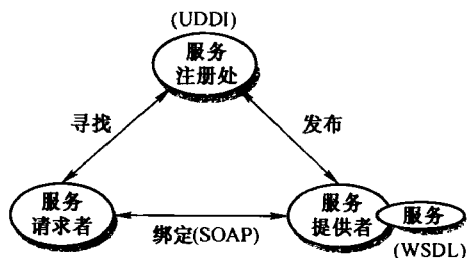


图 19-1 面向服务的体系结构

的 SOA 标准化过程。所有主要硬件和软件公司都接受这些标准。结果是，面向服务的体系结构没有受到不兼容性的困扰，而在技术革新过程中，在不同厂商维护他们各自的技术版权的地方，通常都会出现不兼容性问题。图 19-2 给出了已经建立的一些支持 Web 服务的关键标准。由于早期的标准化，有些问题，如第 17 章中讨论的 CBSE 中的多个不兼容组件模型问题，并没有出现在面向服务的系统开发中。

Web 服务协议覆盖了面向服务的体系结构的所有方面：从基本的服务信息交换（SOAP）机制到编程语言标准（WS-BPEL）。这些标准全部基于 XML——一种人和计算机都可识别的标记语言，它允许定义结构化的数据，其中文本用一个有意义的标识符来标记。XML 有一系列的支持技术，例如用于模式定义的 XSD，它用于扩展和处理 XML 描述。Erl（Erl, 2004）提供了一个有关 XML 技术以及它们在 Web 服务中作用的一个很好的概述。

简要地说，面向 Web 服务的体系结构的主要标准有：

1. **SOAP** 这是一个支持服务之间通信的消息交换标准。它定义服务之间消息传递的必需的和可选的组件。

2. **WSDL** Web 服务定义语言（WSDL）是制定服务接口的标准。它给出了服务是如何操作的（操作名、参数、它们的类型）以及必须定义的服务绑定。

3. **WS-BPDL** 这是一个工作流语言的标准，工作流语言用来定义包括多个不同服务的过程程序。19.3 节讨论过程程序的概念。

服务发现标准（UDDI）也被提到过但还没有广泛采用。UDDI（通用描述、发现和集成）标准定义了服务描述的组件，这种组件可用来发现服务是否存在。它们包括的信息主要有：服务提供者，所提供的服务，服务接口的 WSDL 描述的位置，以及业务关系的信息。目的是这一标准将允许公司建立对他们所提供服务的 UDDI 描述的注册表。

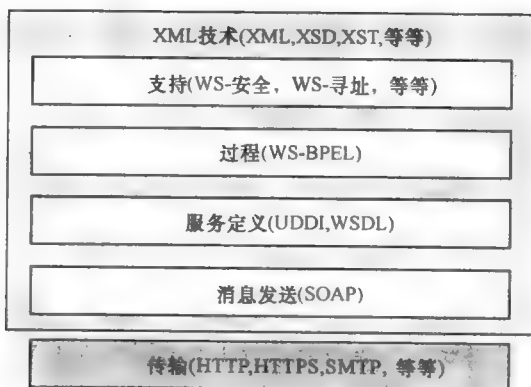


图 19-2 Web 服务标准

许多的公司，比如微软，在 21 世纪初期就建立了 UDDI 注册，但是这些注册应该已经全部关闭。搜索引擎技术的发展使得它们变得多余。服务搜索使用一种标准的搜索引擎来搜索适当评注的 WSDL 描述，这是现在发现外部服务的首选方式。

主要的 SOA 标准受到一系列 SOA 的更专业方面的标准的支持。存在着非常多的支持标准，因为它们希望在不同类型的应用中支持 SOA。这些标准的例子包括：

1. **WS-Reliable Messaging**，是一个确保消息将会传递一次且只一次的消息交换标准。

2. **WS-Security**，是一套支持 Web 服务信息安全性的标准，包括指定信息安全政策定义的标准和覆盖数字签名使用的标准。

3. **WS-Addressing**，定义在一个 SOAP 消息中如何表达地址信息。

4. WS-Transactions, 指定在分布的服务之间的事物应该怎样协调。

Web 服务标准是一个很大的话题, 这里无法详细地讨论它们。有兴趣的读者可以阅读 Ed (2004; 2005) 的书, 以了解这些标准的概况。有关它们的详细描述也可以通过 Web 上的公开文档获得。

当前 Web 服务标准被批评为过于笼统、效率低下的“重量级”标准。实现这些标准需要相当大的过程来创建、传输和解释相关的 XML 信息。由于这一原因, 许多机构, 例如 Amazon, 使用一种更简单、更有效的方法进行服务通信, 它们使用所谓的 RESTful 服务 (Richardson 和 Ruby, 2007)。RESTful 方法支持有效的服务交互, 但是不支持企业级的特征, 比如, WS-Reliability 和 WS-Transactions。Pautasso 等 (2008) 比较了 RESTful 方法和标准的 Web 服务。



RESTful Web 服务

REST (Representational State Transfer) 是一种基于服务器向客户机转移资源描述的体系结构风格。这种风格以 Web 是一整体为基础, 并被作为比 SOAP/WDSL 实现 Web 服务更简单的方法来使用。

RESTful Web 服务是通过它的 URL (Universal Resources identifier) 定义所识别的, 并使用 HTML 协议通信。它响应 HTML 的 GET、PUT、POST 和 DELETE 方法并向客户机返回资源的表示。简单地讲, POST 表示建立, GET 表示读取, PUT 表示更新, DELETE 表示删除。

RESTful 服务要比所谓的“big Web service”有更低的开销, 并且被许多机构使用来实现基于服务的系统, 但它们的系统是不依赖于外部提供的服务的。

<http://www.SoftwareEngineering-9.com/Web/Sevices/REST/>

基于服务构建应用程序允许公司和其他机构进行协作并使用彼此的业务功能。因此, 涉及公司边界间大量信息交换的系统, 比如, 供应链系统, 即一家公司向另一家公司下商品订单, 可以很容易地自动化。如 19.3 节所讨论的, 基于服务的应用的构建, 可以通过连接来自不同提供商的服务, 使用标准编程语言或者专门的工作流语言。

SOA 是松散耦合的体系结构, 服务绑定在执行阶段是可以改变的。这意味着可能会在不同的时间执行着服务的不同的但等价的版本。有些系统完全是使用 Web 服务来构建的, 而其他一些系统是结合了 Web 服务和本地开发的组件。为了说明混合使用服务和组件的应用是如何组织的, 考虑以下情形:

轿车内信息系统驾驶人员提供关于天气、交通状况、本地信息等内容。这是链接到车上无线电装置上的, 这样信息作为信号可以在专设的频道上传输。轿车上配备 GPS 接收装置来发现自己所在的位置, 基于这个位置信息, 系统方位访问一系列信息服务。信息可以使用驾驶员专用语言来传输。

图 19-3 说明了这样一个系统的可能组成。车内软件包括 5 个模块。它们处理与驾驶人员、与报告车辆位置的 GPS 接收装置, 以及与车上的无线电接收装置的通信。Transmitter (传递器) 和 Receiver (接收器) 两模块处理所有与外部设备之间的通信。

车辆与外部移动信息服务通信, 这个外部信息服务又聚集了很多来自其他模块的一些服务, 比如气象服务、交通信息服务和本地设施服务等提供的信息。位于不同地点的不同提供商提供这个服务, 车内系统用发现服务来定位最合适的信息服务并绑定它。移动信息服务也使用发现服务来绑定合适的气象、交通以及设施等服务。服务交换 SOAP 消息 (它包括服务所使用的 GPS 位置信息) 来选择合适的信息。所收集的信息通过一个能将信息语言翻译成驾驶人员的首选语言的服务传回到车内。

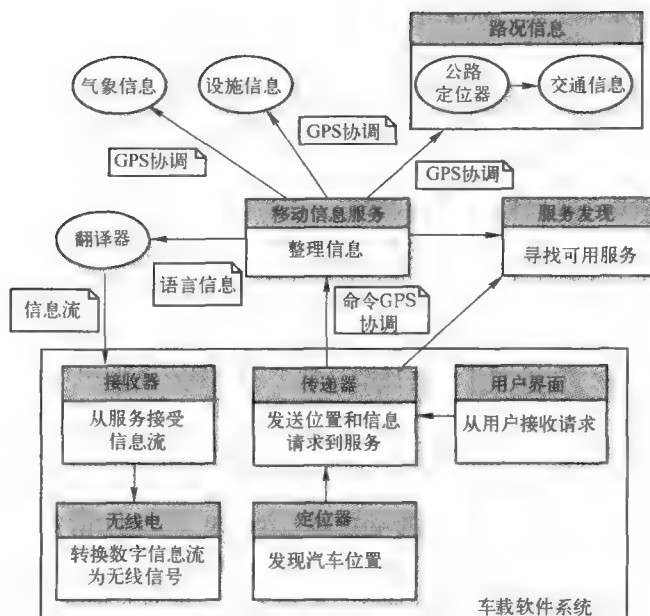


图 19-3 基于服务的车载信息系统

这个例子很好地说明了面向服务方法的一个重要优势。这就是，在系统编程阶段和部署阶段无需决定使用哪个服务提供商，也无需知道需要访问什么特殊的服务。当汽车随处行驶的时候，车载软件使用服务发现服务来寻找最为合适的信息服务并绑定它。由于使用了翻译服务，它能够跨越边界而让不懂当地语言的驾驶人员了解到当地的信息。



面向服务和面向组件的软件工程

服务和组件显然有很多共同之处。它们都是可复用的元素，如第 17 章所述，可以把组件认为是服务的提供者。不过，服务和组件之间以及软件工程面向服务的方法和面向组件的方法之间有很重要的不同之处。

[http://www. SoftwareEngineering-9. com/ Web/ Services/ Comps. html](http://www.SoftwareEngineering-9.com/Web/Services/Comps.html)

软件工程的面向服务方法是一种新的软件工程范式，在作者看来，是与面向对象软件工程同样的一个软件工程的重要进展。这种范式的转变将加速“云计算”（Carr, 2009）的发展，在云计算中服务是由如 Google 和 Amazon 这样主要供应商主办的一个公用计算基础设施所提供的。这已经并将继续对系统产品和业务流程产生深远影响。Newcomer 和 Lomow（2005），在他们关于 SOA 的书上总结了面向服务的方法的潜力：

“由于关键技术的集成和 Web 服务普遍采用的驱动，面向服务的企业承诺极大地提高企业的灵活性，加快新产品和服务推向市场的步伐，降低 IT 的成本并改善运作的效率。”

我们仍处在通过 Web 访问的面向服务应用开发的初期阶段。但是，随着 Google Apps 和 Salesforce.com 等系统的出现，我们已经看到在软件的实施和部署方式上的重大变化。面向服务的方法同时也在应用和实施层面上意味着 Web 正从一个信息存储变成一个系统的实施平台。

19.1 服务作为可复用的组件

在第 17 章中，介绍了基于组件的软件工程（CBSE），在那里软件系统是通过软件组件的

组合来构建的，组件都是基于某一标准组件模型的。服务是软件组件的自然发展，组件模型本质上是一组与 Web 服务关联的标准。服务因此可以被定义为：

松散耦合的、封装了离散功能的可复用软件组件，它可以是分布的，且根据标题来访问。Web 服务是这样一个服务，使用标准的 Internet 和基于 XML 的协议进行访问。

如在 CBSE 中所定义的，服务和软件组件之间的一个重要的区别是，服务应该总是独立的和松散耦合的。即它们应该总是以相同的方式运行，而与它们的执行环境无关。其接口是提供服务功能访问的“提供”接口。服务规定为在不同的上下文中独立和可用的。因此，它们没有“要求”接口，而在 CBSE 中这是定义那些必须出现的其他系统组件的一种接口。

服务通过交换以 XML 表示的消息来通信，这些消息通常使用标准的 Internet 传输协议（如 HTTP 和 TCP/IP）来分发。第 18 章中已经提到这种基于消息方法的组件通信。服务定义它对另一个服务的需要，通过在消息中陈述需求并发送给相应的服务。接收方解析消息，执行计算，完成后发送一个回复，作为消息，给请求的服务。提出请求的服务于是解析应答消息，提取所需要的信息。与软件组件不同，服务不使用远程过程或方法调用来访问与其他服务关联的功能。

当你想要使用 Web 服务的时候，你需要知道服务在哪里（它的 URI）以及接口细节。这些会在服务描述中说明，服务描述用 XML 语言书写，被称为 WSDL（Web 服务描述语言）。WSDL 描述定义了有关 Web 服务的 3 件事：服务做什么，如何通信，以及在哪里能找到它：

1. WSDL 文档的“what”部分，称为接口，指定服务所支持的操作，并且定义服务发送和接收的消息的格式。

2. WSDL 文档的“how”部分，称为一个绑定，把抽象接口映射到一组具体的协议上。绑定指定了如何与一个 Web 服务通信的技术细节。

3. WSDL 文档的“where”部分，描述一个特定的 Web 服务实现的位置（它的端点）。

WSDL 概念模型（见图 19-4），给出了服务描述的元素。其中每个元素用 XML 表达并且可以在单独的文件中完成。这些部分是：

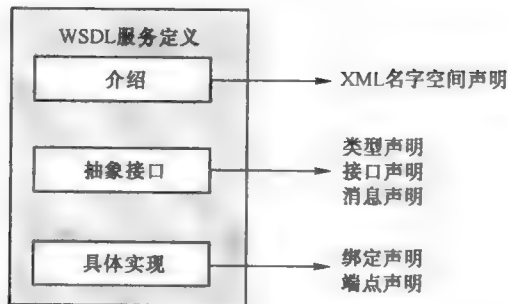


图 19-4 WSDL 描述的组织

1. 介绍性部分，通常，定义 XML 所使用的名字空间，并且可能包含一个提供有关服务的额外信息的文档部分。

2. 可选部分，描述服务交换的消息所使用的一些类型。

3. 服务接口的描述部分，即服务为其他的服务或用户提供的操作。

4. 描述服务所处理的输入和输出消息部分。

5. 服务所使用的绑定的描述部分（例如发送和接收消息的消息传送协议。）默认是 SOAP，但也可以指定其他绑定。绑定说明如何将与服务关联的输入和输出消息封装在消息中，并且指定所使用的通信协议。绑定也可以指定支持信息，诸如安全证书或事务标识符将包括在内。

6. 端点描述部分，这是服务的物理位置，表示为统一资源标识符（URI）——Internet 上能

访问的资源地址。

用 XML 书写的完整服务描述，读起来是冗长、琐碎且枯燥的。它们通常包含 XML 名字空间的定义，这些都是合适的名称。名字空间标识符可以位于任何 XML 标识符之前，使得具有相同名称的标识符可以定义在两个不同的 XML 描述部分中。读者不需要了解名字空间的细节来明白这里的例子。读者只需要知道，名称前面可以有一个名字空间标识符作为前缀，namespace: name 成对出现且唯一的。

WSDL 描述现在很少手工书写，描述中的大多数信息都是自动生成的。读者不需要知道描述的具体细节来理解 WSDL 的原理，所以，将集中解释抽象接口的描述。这是 WSDL 描述部分，相当于软件组件的“提供”接口。图 19-5 给出一个简单服务的部分接口，给定一个日期和一个地点，指定一个国家内的城市，返回在那个地点和时间所记录的最高和最低的温度。输入的消息也指定这些温度是以摄氏温度还是以华氏温度返回。

Define some of the types used. Assume that the namespace prefix 'ws' refers to the namespace URI for XML schemas and the namespace prefix associated with this definition is weathns.

```
<types>
  <xs:schema targetNamespace = "http://.../weathns"
    xmlns: weathns = "http://.../weathns" >
    <xs:element name = "PlaceAndDate" type = "pdrec" />
    <xs:element name = "MaxMinTemp" type = "mmtrec" />
    <xs:element name = "InDataFault" type = "errmess" />

    <xs:complexType name = "pdrec"
      <xs:sequence>
        <xs:element name = "town" type = "xs:string"/>
        <xs:element name = "country" type = "xs:string"/>
        <xs:element name = "day" type = "xs:date" />
      </xs:sequence>
    </xs:complexType>
```

Definitions of MaxMinType and InDataFault here

```
</schema>
```

```
</types>
```

Now define the interface and its operations. In this case, there is only a single operation to return maximum and minimum temperatures.

```
<interface name = "weatherInfo" >
  <operation name = "getMaxMinTemps" pattern = "wsdl: in-out">
    <input messageLabel = "In" element = "weathns: PlaceAndDate" />
    <output messageLabel = "Out" element = "weathns:MaxMinTemp" />
    <outfault messageLabel = "Out" element = "weathns:InDataFault" />
  </operation>
</interface>
```

图 19-5 一个 Web 服务的部分 WSDL 描述

在图 19-6 中，描述的第一部分显示了在服务描述中所使用的元素和类型定义部分，定义了元素 PlaceAndDate、MaxMinTemp 和 InDataFault。这里只给出 PlaceAndDate 的描述，可以看出它是包含 3 个域（城镇、国家和日期）的一个记录。用类似的方式可以定义 MaxMinTemp 和 InDataFault。

描述的第二部分给出如何定义服务接口。在此例子中，服务 weatherInfo 有唯一操作，尽管没有可定义的操作个数的限制。weatherInfo 操作有一个相关的输入-输出模式，获取一个输入消息，产生一个输出消息。WSDL2.0 描述允许多个不同的消息交换模式，例如 in-only、in-out、out-only、in-optional-out、out-in 等。输入和输出消息，即那些在类型域部分中较早给出的声明，此时完成定义。

WSDL 的主要问题是服务接口定义不包含任何服务语义或它的非功能特性的信息, 例如性能和可依赖性。它仅仅是一个服务签名的描述 (例如操作和其参数)。准备使用该服务的程序员不得不搞清楚服务到底能做什么, 以及它的输入和输出消息的不同域的意思。性能和可依赖性必须由服务的实验来验证。直观的名称和服务文档会帮助读者理解服务提供的功能, 但读者仍有可能对服务产生很多误解。

19.2 服务工程

服务工程是开发服务的过程, 这种服务是面向服务的应用开发中可复用的。它和组件工程非常类似。服务工程师必须确保服务代表可复用的抽象, 能用于不同系统的抽象。他们要设计开发与此抽象关联的有用的一些功能, 而且确保服务是健壮的和可靠的。他们必须为服务提供文档, 以便服务能被需要的用户所发现和了解。

在服务工程过程中有如下 3 个逻辑阶段, 如图 19-6 所示。

1. 可选服务识别, 在此我们识别那些需要实现的服务, 并定义服务需求。
2. 服务设计, 在此我们设计逻辑和 WSDL 的服务接口。
3. 服务实现和部署, 在此我们实现并测试服务, 使之可用。

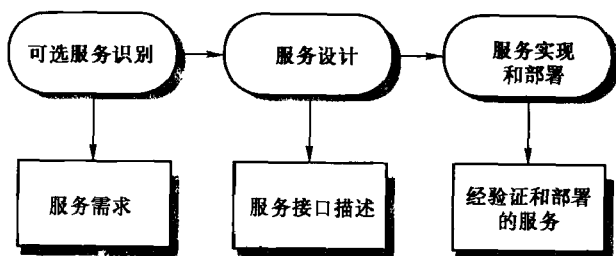


图 19-6 服务工程过程

如第 16 章中提到的, 可复用组件的开发可能是在已经实现了的并在应用程序中使用的现有组件的基础上开始的。服务同样如此, 此过程的起点往往是一个现有的服务或者要转化为服务的组件。在这种情况下, 设计过程涉及归纳现有组件, 删除应用程序特定的功能。实现意味着通过添加服务接口改写原来的组件并实现所需要的泛化。

19.2.1 可选服务的识别

面向服务计算的基本理念是服务应该支持业务过程。由于每个机构都有很多的过程, 因此存在许多可能的服务可以加以实现。可选服务识别需要理解和分析机构的业务过程, 来决定哪些可复用服务可能需要实现以支持这些过程。

Erl 建议可以定义三种基本服务类型:

1. **实用服务** 这些服务实现某些一般性的功能, 可被用于不同的业务过程。一个实用服务的例子是货币转换服务, 通过访问它可以计算一种货币 (例如美元) 对另外一种货币 (例如欧元) 的兑换。

2. **业务服务** 这些服务是与特殊业务功能相关的服务。大学里的业务功能的例子是学生为一门课程注册登记。

3. **协同或过程服务** 这些服务是为支持更一般的业务过程, 这些业务过程总是包含不同的角色和活动。公司里的协同服务的例子是订货服务, 允许完成一个包含厂商、产品以及付款方式的订单。

Erl 也建议将服务看做是面向任务的或面向实体的。面向任务的服务是与某项活动关联的, 而面向实体的服务就像对象, 即与某个业务实体关联, 这样的业务实体的例子是, 一张求职申请

表。图 19-7 给出了一些面向任务的和面向实体的服务的例子。实用或业务服务可能是面向实体的或是面向任务的，但协同服务总是面向任务的。

在可选服务识别阶段的目标应该是找出那些逻辑上相关的、独立的且可复用的服务。Edl 的分类在这方面是有帮助的。它给出了如何通过对业务实体和业务活动的观察来发现可复用的服务。然而，识别可选服务有时是很困难的，因为你不得不想象这项服务将是如何使用的。你必须考虑所有可能的候选项，然后通过一系列关于它们的问题来分析是否像是有用的服务。能帮助你发现可复用的服务的问题有：

1. 对于一个面向实体的服务，它是与一单个用于不同业务过程的逻辑实体关联的吗？通常情况下在必须支持的实体上都执行哪些操作？
2. 对于一个面向任务的服务，该任务是在机构中由不同的人执行的吗？当提供一单个支持服务时要发生不可避免的标准化问题，他们愿意接受吗？
3. 服务是独立的吗？也就是说，它在多大程度上依赖其他服务的可用性的？
4. 对于它的操作，服务必须维护状态吗？服务是无状态的，这意味着服务不用维护中间状态。如果需要状态信息，将不得不使用数据库，而这将限制服务的复用性。总的来说，服务传递状态到服务是易于复用的，因为不需要绑定数据库。
5. 服务能被机构外面的客户使用吗？举例来说，一个与目录关联的面向实体的服务可以既在内部访问又可以由外部访问吗？
6. 服务的不同用户可能有不同的非功能性需求吗？如果有，那么就应该实现不止一个版本的服务。

	实 用	业 务	协 同
任 务	货币兑换器 员工定位器	验证申报表格 检查信用等级	过程费用申报 支付外部供应商
实 体	文档风格检查器 Web 表格到 XML 转换器	费用表格 学生申请表	

图 19-7 服务分类

这些问题的答案有助于我们去选择和精练那些将被实现为服务的抽象。然而，决定哪个是最好的服务并没有公式化的方法，因此服务识别是一个基于技术和经验的过程。

可用服务选择过程的输出是一组找到的服务以及相关的需求。功能性服务需求需要定义服务应该做什么。非功能性需求需要定义服务的信息安全性、性能和可用性需求。

为了帮助读者理解可选服务的识别和实现，考虑下面的例子：

一家卖计算机设备的大公司，已经为一些客户的核准配置安排了特殊价格。为了方便自动订购，公司希望制作一个目录服务，允许客户选择他们需要的设备。与消费者目录不同，订单不是直接地通过目录接口下达。相反，商品是通过每家公司的基于 Web 的采购系统下订单的，每家公司以 Web 服务来访问目录。绝大多数公司有它们自己的预算流程和订单批复流程，在下订单时必须遵循他们自己的流程。

目录服务是支持业务操作的面向实体服务的一个例子。目录服务的功能性需求如下：

1. 目录的特定版本将提供给每家用户公司。它包括由客户公司的员工所订购的配置和设备以及目录项的议定价格。
2. 目录应该允许客户公司职员下载目录版本以便脱机浏览。
3. 目录应该允许用户比较多达 6 个目录项的描述和价格。
4. 目录应该为用户提供浏览和搜索工具。

5. 目录的用户应该能够根据某一特定目录项的代码发现可预期的交付日期。

6. 目录的用户应该能够下“虚订单”，即所需的项可以为他们保留 48 小时。虚订单必须通过采购系统下达的一个真实订单所确定。确认消息必须在虚订单的 48 小时内收到。

除了这些功能性需求之外，目录还有许多的非功能性需求：

- 1. 访问目录服务的权限将限制为认可的机构职员。
- 2. 提供给一客户的价格和配置信息将是保密的，对任何其他客户来说将是不可见的。
- 3. 目录在从格林尼治标准时间 0700 到格林尼治标准时间 1100 都将可用且不间断。
- 4. 目录服务应该在峰值负载能够每秒处理高达 10 个请求。

注意这里没有有关目录服务响应时间的非功能性需求。这依赖于目录的大小和预计的并发用户的多少。因为这不是一个时间要求极高的服务，所以不需要在这个阶段指定它。

19.2.2 服务接口设计

一旦选择了可选服务，服务工程过程的下一个阶段就是设计服务接口。这包括定义与服务关联的操作以及它们的参数。也需要仔细地考虑服务操作和消息的设计，目的是使得完成服务请求一定要发生的消息交换的次数最小。必须确保在一个消息中尽可能多地携带所要传递给服务的信息，而不采用同步的服务交互。

应该记住服务是无状态的，管理特定服务应用状态是服务用户的职责，而非服务本身的责任。因此我们可能需要在服务之间通过输入和输出消息传递状态信息。

服务接口设计有 3 个阶段：

- 1. 逻辑接口设计，找出与服务关联的操作、这些操作的输入和输出，以及与这些操作关联的异常。
- 2. 消息设计，设计由服务发送和接收的消息结构。
- 3. WSDL 开发，用 WSDL 语言将逻辑设计和消息设计翻译成抽象接口描述。

第一阶段，逻辑接口设计，从服务需求开始，定义操作名称和参数。在这个阶段，也要定义当一个服务操作被调用时可能出现的异常。图 19-8 和图 19-9 给出了实现需求的操作，每个目录操作的输入、输出和异常。在这个阶段，不需要详细描述它们，因为将在设计过程的下一个阶段添加细节内容。

操 作	描 述
MakeCatalogue	为特殊客户创建一个目录的特殊版本。包括一个可选参数来创建目录的一个可下载的 PDF 版本
Compare	最多可提供 4 个目录项的 6 个特性以进行比较（例如，价格、尺寸、处理器速度等）
Lookup	显示关联于特殊目录项的所有数据
Search	此操作接受逻辑表达式，根据此表达式搜索目录。显示能匹配表达式的所有项的一个列表
CheckDelivery	如果在某天预订的话，返回此项的预计的交付日期
MakeVirtualOrder	保存将由客户订购的项的数目，并为客户自己的采购系统提供各项的信息

图 19-8 目录服务操作的功能性描述

定义异常和异常如何传达给服务用户是特别重要的。服务工程师不知道他们的服务将会被如何使用，且假定服务用户将会完全理解服务描述通常是不明智的。输入消息可能是不正确的，所以应该定义异常向服务客户报告不正确的输入。在可复用的组件开发中将所有异常处理交给组件的用户通常是个好的做法。服务开发者不应该在如何处理异常上强加自己的观点。

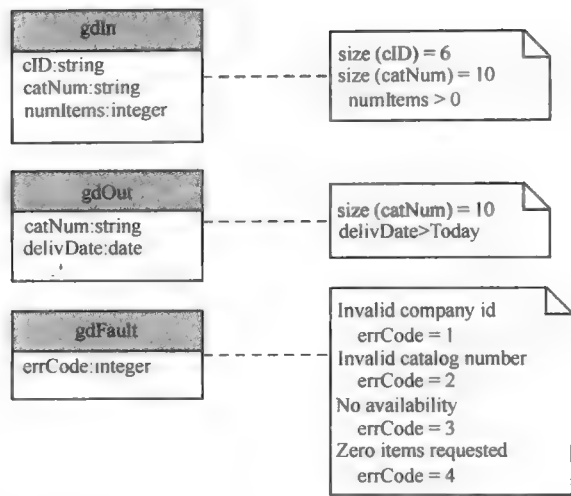


图 19-9 目录接口设计

一旦建立了有关服务应该做什么的非正式逻辑描述，下一个阶段就是定义输入和输出消息的结构以及在这些消息中所使用的类型。XML 不适合在这个阶段中使用。作者认为将消息表示为对象用 UML 或者编程语言（例如，Java）来表示更好。它们能人工地或自动地转换为 XML。图 19-10 显示了目录服务中的 getDelivery 操作的输入和输出消息的结构。

操 作	输 入	输 出	异 常
MakeCatalogue	mcIn 公司 id PDF - 标志	mcOut 该公司目录的 URL	mcFault 无效的公司 id
Compare	compIn 公司 id 入口属性（最多 6 个） 目录数目（最多 4 个）	compOut 显示比较表页面的 URL	compFault 无效的公司 id 无效的目录数 未知的属性
Lookup	lookIn 公司 id 目录数目	lookOut 有项信息页面的 URL	lookFault 无效公司 id 无效目录数目
Search	searchIn 公司 id 搜索字符串	searchOut 有搜索结果页面的 URL	searchFault 无效公司 id 格式不好的搜索字符串
CheckDelivery	gdIn 公司 id 目录数目 请求的项数	gdOut 目录数目 期待的交付日期	gdFault 无效公司 id 无效目录数目 无可用性 零请求项
PlaceOrder	poIn 公司 id 请求的项数 目录数目	poOut 目录数目 请求的项数 预计交付日期 单价估计 总价格估计	poFault 无效公司 id 无效目录数目 零请求项

图 19-10 输入、输出消息的 UML 定义

注意这里是如何把细节加入到描述中的——通过用约束注解 UML 图表。它们定义了表示公司和目录项的字符串的长度，指定项目数必须大于 0，交付必须在当前日期之后。注解也显示了与每个可能的故障关联的错误代码。

服务设计过程的最后阶段是将服务接口设计翻译成 WSDL。如前面所讨论的，WSDL 表示很长且很详细，因此，如果在这个阶段你手工完成这些就很容易出错。不过，大部分的支持面向服务开发的编程环境（比如 ECLIPSE 环境）包含能将逻辑接口描述翻译成它对应的 WSDL 表示的工具。

19.2.3 服务实现和部署

一旦找到了可选服务并且设计了它们的接口，服务工程过程的最后阶段就是服务实现。实现可能涉及使用某个标准的编程语言（例如 Java 或 C#）编写服务程序。这两种语言现在都包括对服务开发广泛支持的库。

另外一种做法是，服务的开发可以通过向现有组件或遗留系统（如稍后将要讨论的）添加服务接口的办法实现。这意味着已经证明了是有用的软件资产能被更加广泛地利用。对于遗留系统的情形，它可能意味着系统功能能被新的应用访问。也可以通过定义现有服务的组合来开发新的服务。19.3 节讨论这种开发服务的方法。

服务一经实现，在部署它之前必须通过测试。这包括检查和划分服务输入（如第 18 章所讨论的），创建反映这些输入组合的输入消息，然后检查输出是否是预期的。在测试中应该总是去尝试产生异常来检查服务能够应付无效输入。测试工具都能允许对服务的检查和测试，并能从 WSDL 描述生成测试。然而，这些只能测试服务接口与 WSDL 的一致性。它们不能测试服务的功能行为。

服务部署是过程的最后阶段，包括在 Web 服务器上部署此服务。绝大多数服务器软件使这一步变得非常简单。你只需在特定目录下安装包含可执行的服务的文件。然后它会自动变得可用。如果希望服务是公用的，必须为服务的外部用户提供信息。这些信息帮助潜在的外部用户明确该服务能否满足他们的需求以及是否值得信任，而对于服务提供商，能够帮助他们安全可靠地提供服务。服务描述中可能包含的信息如下：

1. 有关你的企业的信息、联系方式等。这对用户信任来说是至关重要的。服务的用户必须确信服务将不会表现出恶意的行为。有关服务提供者的信息能让用户在商业信息机构中去检查他们的资格证书。
2. 服务提供的功能的非正式描述。它帮助潜在用户决定服务是否是他们想要的。然而，功能描述使用的是自然语言，因此对服务做什么的描述，它不是无二义的语义描述。
3. 接口类型和语义的详细描述。
4. 订阅信息，允许用户注册以获取有关服务更新的信息。

如前面所述，服务描述的一个一般问题是，服务的功能行为是通过自然语言描述非正式给出的。自然语言描述便于阅读，但是容易引起误解。为了解决这个问题，有一支活跃的研究团体正在研究服务语义如何来定义。语义描述最有前途的方法是基于本体论的描述，既描述中的术语的特定含义是在本体中定义的。本体是一个标准化术语使用方法的方法，并且定义不同术语之间的关系。他们正在越来越多地用于帮助指定自然语言的语义。人们开发了一种被称为 OWL-S 的语言用于描述 Web 服务本体（OWL_Services_Coalition, 2003）。

19.2.4 遗留系统服务

遗留系统是指一个机构使用的旧的软件系统。遗留系统通常依赖过时的技术但是对于业务

仍然至关重要。重写或替换这些系统可能不符合成本效益，许多机构想用更现代的系统与遗留系统结合在一起。服务最重要的一种使用就是为遗留系统实现“包装”，遗留系统提供对系统功能和数据的访问。这些系统于是能通过 Web 访问，并与其他应用集成。

举例说明这一点，设想一家大公司维护着其设备库存和相关设备保养的一个数据库，这个数据库跟踪设备的维护和修理。它跟踪在不同设备上的诸多信息，包括：产生了的维护请求，安排的哪些常规维护，维护是在什么时间完成的，执行维护花费了多少时间，等等。此遗留系统最初是用来为维护人员产生日常的维护清单的，但是随着时间的推移，新的功能不断被添加进来。它们提供有关维护每件设备的花费的数据和帮助外部承包商对执行的维护工作进行估价的信息。系统作为客户机-服务器系统运行，专用客户端软件装在 PC 上运行。

公司现在希望能让维护人员从便携终端实时访问此系统。维护人员直接用维护所花费的时间和资源去更新系统，并通过查询系统来发现他们的下一步维护工作。除此之外，呼叫中心人员要求访问系统以记录维护请求和检查他们的状态。

增强系统以支持这些需求事实上是不可能的，所以公司决定为维护人员和呼叫中心人员提供新的应用。这些应用依赖遗留系统，遗留系统将被用作实现多个服务的基础。在图 19-11 中说明，这里用了 UML 模式来说明一个服务。新应用仅与这些服务交换消息来访问遗留系统的功能。

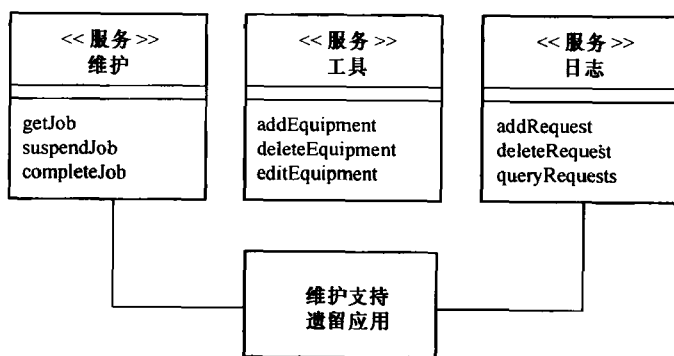


图 19-11 对遗留系统提供访问的服务

所提供的一些服务如下：

1. **维护服务** 包括的操作有：根据施工号码、优先级和地理位置来检索一项维护工作；上传已经执行的维护到数据库。它也支持将已经启动但是未完成的维护工作暂停和重新开始的操作。
2. **工具服务** 这包括添加和删除新设备的操作，修改数据库中与设备关联的信息的操作。
3. **日志服务** 包括为服务添加一个新请求、删除维护请求和查询未完成请求的状态。

注意现有的遗留系统不是简单地被表示为一个服务。更确切地说，为访问遗留系统所开发的服务是关联的，每一个支持单一方面的功能。这不但降低了它们的复杂性而且使它们在其他应用中更容易理解和复用。

19.3 使用服务的软件开发

使用服务的软件开发大概是基于这样的思想：组合并配置服务来创建新的复合服务。这些复合服务可以与一个在浏览器上实现的用户界面集成来创建一个 Web 应用，或者可以被当做组件用于某个其他服务组合。组合中所包含的服务可能是专门为一特殊应用开发的，可能是公司内部开发的业务服务，或者可能是来自某个外部提供者的服务。

许多公司目前正把他们的企业应用程序转换为面向服务的系统，面向服务的系统的基本应用程序是服务而不是组件。这开启了在公司内部更加广泛复用的可能性。下一个阶段将会是开发机构间的相互信任的供应商之间的应用，它们彼此使用对方的服务。面向服务体系结构的远景目标的最终实现将依赖于“服务市场”的开发，SOA 的长远梦想将依赖于“服务市场”的发展，即服务是从外部的提供商购买的。

服务组合是一个集成过程，可以用来集成分离的业务过程从而能提供更加广泛的功能。比方说一家航空公司希望为旅客提供一个完整的假期计划。除了预订他们的班机外，旅客也能预订在他们所希望的地点的旅馆，安排租车或预订出租车到机场接机，浏览旅行指南以及预约当地名胜的游览。为了创建此应用，航空公司将它的订票服务、旅馆预订代理所提供的服务、汽车租赁和出租车公司的服务，以及当地名胜旅游提供者所提供的预约服务组合在一起。最终结果是一个服务，它集成了这些来自于不同提供者的服务。

你可以思考是否此过程是如图 19-12 所示的一些分离步骤的序列。信息从一步传递到下一步，例如，汽车租赁公司被告知班机预定到达的时间。步骤序列被称为一个工作流——一组在时间上有顺序的活动，每个活动执行工作的某个部分。工作流是业务过程的一个模型（例如，列出在达到一个对于业务特别重要的特别目标过程中所涉及的所有步骤）。在此情景中，业务过程是航空公司提供的假期预订服务。

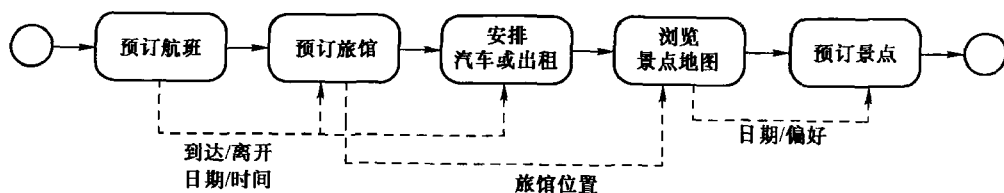


图 19-12 一揽子假期工作流

工作流是一个简单的思想，上述的假期预订的假定情节看起来是很简单的。事实上，服务组合要比这个简单模型所暗示的要复杂得多。举例来说，你必须考虑服务失效的可能性，并加入一些机制来处理这些失败。你也需要考虑应用用户会提出的非预期的要求。举例来说，假定一位旅客是残疾人，需要租用一个轮椅到飞机场。这将需要实现并组合额外的服务，还需要向工作流中添加额外的步骤。

我们必须能够应付这样的局面：工作流不得不改变，因为某一个服务的正常执行总是导致与其他服务执行的不兼容。举例来说，假定所预订的一架班机将在 6 月 1 日离开在 6 月 7 日返回。于是工作流进入旅馆预订阶段。然而，旅游胜地直到 6 月 2 日一直都在召开一个重要会议，因此旅馆没有可用的房间。旅馆预订服务报告此不可用性。这不是一个失败，缺乏可用性是一种常见的情形。于是我们不得不“撤销”班机预订并将有关缺乏可用性的信息返回给用户。然后他或她就会决定是否改变他们的日程或他们的度假地。在工作流术语中，这叫做“补偿动作”。补偿动作用来撤销一些动作，这些动作是已经完成的但是因为后续工作流活动的结果必须改变。

通过复用现有的服务的新服务设计过程，本质上是使用复用的软件设计过程（见图 19-13）。使用复用的设计不可避免要在需求上做出妥协。系统“理想化”的需求就不得不修改以迎合实际上可用的服务，利用这些可用服务，成本就能在预算之内，且服务的品质是可以接受的。

图 19-13 中，在通过组合构建服务的过程中已经给出了 6 个关键阶段：

1. 表达概略工作流 在此服务设计的起始阶段，使用复合服务的需求作为创建“理想”服

务设计的基础。应该在此阶段创建一个相当抽象的设计,以便当对可用服务了解得更多的时候能够添加细节。

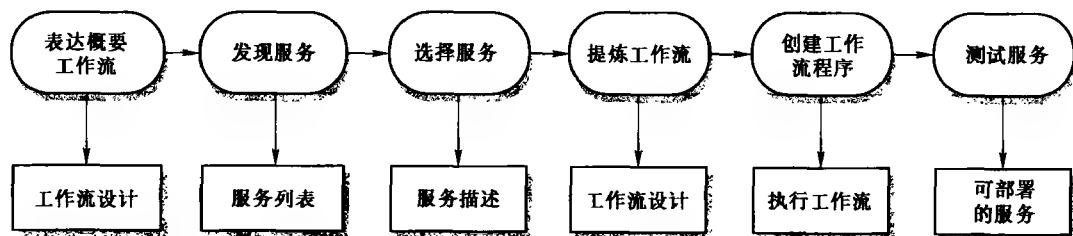


图 19-13 通过组合的服务构造

2. **发现服务** 在过程此阶段,搜索服务注册处或目录来发现存在什么服务、谁提供这些服务以及服务提供的一些细节信息。

3. **选择可能的服务** 从已经发现的可能的服务候选集合中,选择能实现 workflow 活动的可能服务。选择标准肯定要包括所提供的服务的功能,也可能包含所提供服务的成本和服务品质(响应、有用性等)。可以选择许多功能等效的服务,根据其成本和质量将它们绑定到某个 workflow 活动上。

4. **精炼工作流** 基于已经选择服务的信息,精炼工作流。这包括把细节加入到抽象描述中,以及可能的话,增加或删除 workflow 活动。然后可能要重复服务发现和选择阶段。一旦选择了一组稳定的服务,并建立起了最终的工作流,就转移到过程的下一个阶段。

5. **创建工作流程序** 在此阶段,将抽象工作流设计转换为一个可执行程序并定义服务的接口。服务实现可以使用常用的编程语言,例如 Java 或 C#,或 workflow 语言,如 WS-BPEL。如前面所讨论的,服务接口描述应该用 WSDL 来写。这一阶段也可能包含基于 Web 的用户界面的创建,从而允许从 Web 浏览器访问新的服务。

6. **测试完成的服务或应用** 在使用外部服务的情形,对完工的复合服务的测试过程比组件测试更为复杂。19.3.2 节将讨论测试问题。

在本章的剩余部分,将重点放在 workflow 设计和测试上。实际上,服务发现看起来并不是一个主要问题。实际情况仍然是绝大多数服务复用是机构内的,在组织内通过使用内部注册处和软件工程师之间的非正式交流,这些服务得以发现。标准的搜索引擎可以用来发现公共的可用服务。

19.3.1 工作流设计和实现

工作流设计包括分析现有的或计划中的业务过程来理解所发生的不同活动以及这些活动是怎么交换信息的,接着用 workflow 设计符号定义新的业务流程。工作流设计列出了执行过程中的各个阶段和不同过程阶段之间所传递的信息。然而,已有的过程可能是非正式的且是依赖于过程中人的技术和能力,而并没有一个“规范”的工作方式或过程定义。在这种情况下,我们就不得不使用当前过程的知识来设计工作流,使之达到的同样的目标。

工作流表示的是业务过程模型,且通常使用图形形式来描述,例如 UML 活动图,或 BPMN,即业务过程建模符号(Business Process Modeling Notation)系统(White, 2004a; White 和 Miers, 2008)。它们提供类似的功能(White, 2004b)。作者认为 BPMN 和 UML 活动图以后可能会整合到一起,工作流模型标准的制定将会是基于这种整合后的语言的。本章中的例子使用 BPMN 描述。

BPMN 是一种相当容易理解的图形语言。人们定义了一种映射,将此语言翻译为较低级的、基于 XML 的描述——WS-BPEL 描述。因此 BPMN 与一系列在图 19-2 中所示的 Web 服务标准是一致的。

图 19-14 是 BPMN 模型的一个简单例子，这是前面的假期计划脚本的部分内容。模型给出了旅馆预订的简化 workflows，假定存在一个 Hotels 服务，其关联操作称为 GetRequirements、CheckAvailability、ReserveRooms、NoAvailability、ConfirmReservation 和 CancelReservation。此过程包括得到来自客户的需求，然后检查是否有空余房间可用，如果房间是可用的，按照所要求的日期预订房间。

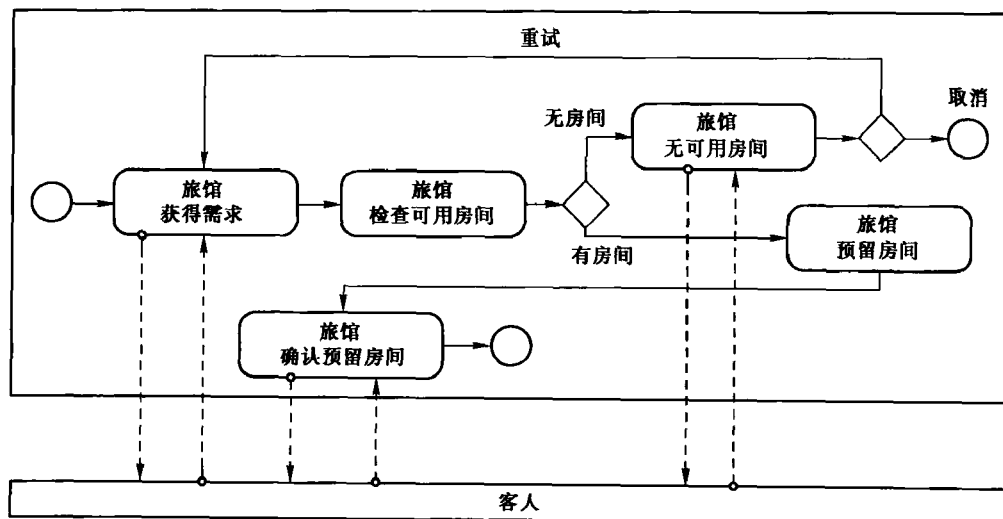


图 19-14 宾馆预定 workflow 片段

此模型导入了 BPMN 的一些用来创建工作流模型的核心概念：

1. 圆角矩形表示活动。一个活动能被一个人或一个自动化服务执行。
2. 圆圈表示事件。一个事件是在一个业务过程当中发生的某事。简单圆圈用来表示一个开始事件，较黑的圆圈表示一个结束事件。双层圆圈（没有给出）用来表示一个中间事件。事件可以是时钟事件，因而允许工作流定期地或超时地执行。
3. 菱形用来表示一个通道。通道是过程中的一个阶段，在此做出某个选择。举例来说，在图 19-14 中，有一个选择是根据房间是否可用而做出的。
4. 实线箭头用来表示活动序列；虚线箭头表示活动之间的消息流，在图 19-14 中，这些消息在旅馆预订服务与客户之间传递。

这些主要特征足够描述大多数工作流的本质。然而，BPMN 还包含许多另外的特征，这里不再过多描述。这些特征将信息添加到业务过程描述中，使之能够自动被翻译为一种可执行的服务。因此，用 BPMN 描述的服务组合成的 Web 服务可以从业务过程模型中直接生成。

图 19-14 显示的是在某机构中使用的过程，这是一个提供预订服务的公司。然而，面向服务方法的主要好处在于它支持机构间计算。这意味着整个计算涉及不同公司的服务。这可以用 BPMN 表示出来，通过为每个参与交互的机构开发单独的工作流完成的。

为了阐明这一点，采用另外一个例子，该例子与高性能计算有关。面向服务的方法的提出允许共享诸如高性能计算机这样的资源。在本例中，假定一个矢量处理计算机（一个能在值数组上执行并行计算的机器）作为一个服务（VectorProcService）由一研究实验室提供。它通过另外一个被称为 SetupComputation 的服务进行访问。这些服务以及它们的交互如图 19-15 所示。

在此例中，SetupComputation 服务的工作流请求访问一矢量处理机，如果有处理机是可用的，就建立所需要的计算并下载数据到正在处理的服务。一旦计算完成，结果便被存储在本地计算机上。VectorProcService 的工作流检查处理机是否可用，为计算分配资源，初始化系统，执行计

依赖于对软件源代码的分析。然而，当服务由一外部提供者所提供时，服务实现的源代码是不可得的。因此基于服务的系统的测试不能使用公认的基于源代码的技术。

在对服务和组合进行测试时，除了有对服务实现的理解问题外，测试者也可能面对更大的困难：

1. 外部服务是受控于服务提供者而非服务的用户。服务提供者随时可能撤销这些服务或改变它们，而这些都将使先前的所有应用测试失效。这些问题在软件组件中是通过维护组件的不同版本得到解决的。然而现在，还没有提出来任何标准去处理服务版本。

2. 面向服务体系结构的远景目标是服务动态绑定到面向服务的应用。这意味着，一个应用在每次执行时可能不总是使用相同的服务。所以，测试当一个应用被绑定到某个特殊服务的时候可能是成功的，但是不能保证那个服务将在系统的一个实际执行过程中得到使用。

3. 服务的非功能性行为不只依赖于它是如何被正在测试的应用使用的。一个服务在测试期间可能执行得很好，因为它没有在一个很重的负载之下运行。实际中，所观察的服务行为可能是不同的，因为别的用户所提出的要求是不一样的。

4. 服务的支付模型可以使服务测试变得非常昂贵。有不同种可能的支付模型——某些服务可以是免费得到的，有些服务通过注册付款，其他服务是每次使用时付款。如果服务是免费的，那么服务提供者不希望服务被正在进行测试的应用加载；如果需要注册，那么服务用户可能在测试服务之前不情愿同意它的注册条款；同样，如果使用服务是基于每次使用付款的，服务的用户可能会发现测试成本令人望而却步。

5. 前面已经讨论了补偿动作的概念，当某个异常发生而先前做出的承诺（例如一次班机预订）不得不撤销的时候，补偿动作被调用。对这样的动作进行测试存在一个问题，因为它们可能与其他服务失败有关。确保这些服务在测试过程内真的失效可能非常困难。

当使用的是外部服务时，这些问题就特别敏感。当服务来自于同一公司或对伙伴公司所提供的服务信任时，解决这些问题就比较轻。在这种情况下，源代码是可以得到的，可以用它们来指导测试过程，对服务的支付也不是什么问题。解决这些测试问题，提出对测试面向服务应用的准则、工具和技术仍然是一个重要的研究课题。

要点

- 面向服务的体系结构是可复用软件工程的一种方法，可复用的标准化的服务是应用系统的基本积木块。
- 服务接口是用一种被称为 WSDL 的基于 XML 的语言定义的。一个 WSDL 描述包含：对接口类型和操作的定义，由服务所使用的绑定协议，以及服务的位置。
- 服务可以被分为：提供某一通用目的的实用服务、实现部分业务过程的业务服务或协调其他服务执行的协同服务。
- 服务工程过程包括为实现找出可选服务、定义服务接口并实现、测试和部署服务。
- 对机构仍然有用的遗留软件系统，可能要定义服务接口。这样遗留系统的功能就可以在其他应用中继续得到使用了。
- 使用服务的软件开发是基于这样的思想：程序的创建是通过组合并配置服务来创建复合服务。
- 业务过程模型定义活动和发生在某个业务过程中的信息交换。业务过程中的活动可以由服务实现，这样业务过程模型代表的的是一个服务组合。

进一步阅读材料

网上有大量的指导材料，涵盖了 Web 服务的所有方面。然而，作者发现 Thomas Erl 所著的

下面两本书是对服务以及服务标准最好的综述和阅读材料。与绝大多数的书不同, Erl 在面向服务计算方面给出了一些有关软件工程问题的讨论。他还写了更专业的关于服务设计和 SOA 模式设计方面的书, 但是这些书一般都是针对有 SOA 实现经验的读者。

《Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services》本书的主要聚焦在作为 SOA 框架的重要的基于 XML 的技术 (SOAP、WSDL、BPEL 等) (T. Erl, Prentice Hall, 2004)。

《Service-Oriented Architecture: Concepts, Technology and Design》这是一本关于面向服务系统工程的更全面的书。这本书与上边那本有一点重叠, 但是 Erl 主要聚焦在讨论面向服务方法是如何在软件过程的各个阶段使用的 (T. Erl, Prentice Hall, 2005)。

《SOA realization: Service design principles》这篇短小的网络文章是对服务设计中应考虑的问题的一个很好的概述 (D. J. N. Atrus, IBM, 2006)。http://www.ibm.com/developerworks/webservices/library/ws-soa-design/。

练习

- 19.1 服务和软件组件之间的最主要区别是什么?
- 19.2 解释为什么面向服务的体系结构应该基于标准。
- 19.3 使用相同的符号, 扩展图 19-5 使包含对 MaxMinType 和 InDataFault 的定义。温度应该表示为整数, 有一个附加字段指示是否为华氏温度或摄氏温度的。InDataFault 应该是一个包含一错误代码的简单类型。
- 19.4 为图 19-7 所示的 Currency Converter (货币兑换) 和 Check credit rating (检查信用等级) 两服务分别给出一接口定义描述。
- 19.5 为图 19-11 所示的服务设计可能的输入和输出消息。可以用 UML 或 XML。
- 19.6 给出两种重要的应用类型, 对于这两种应用类型你会建议不要使用面向服务的体系结构。给出你的理由。
- 19.7 在 19.2.1 节中, 介绍了一个开发目录服务公司的例子, 此目录服务是客户的基于 Web 的采购系统所调用的。用 BPMN 设计一 workflow, 使用这个目录服务查询和下达计算机设备的订单。
- 19.8 解释一下“补偿动作”的含义, 并用一例子说明为什么这些动作可能必须包含在 workflow 中。
- 19.9 对于一揽子假期预订服务这个例子, 设计一个为一批到达机场的乘客预订地面运输的工作流。应该提供给乘客选项使之可以选择出租车还是大巴。你可以假定出租车和汽车租赁公司提供 Web 服务支持预订。
- 19.10 用一个例子, 详细解释为什么对包含补偿动作的服务进行彻底的测试是困难的。

参考书目

Andrews, T., Curbera, F., Golland, Y., Klein, J. and Al., E. (2003). 'Business Process Execution Language for Web Services'. <http://www-128.ibm.com/developerworks/library/ws-bpel/>.

Cabrera, L. F., Copeland, G. and Al., E. 2005. 'Web Services Coordination (WS-Coordination)'. <ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf>.

Carr, N. (2009). *The Big Switch: Rewiring the World from Edison to Google, Reprint edition*. New York: W.W. Norton & Co.

Erl, T. (2004). *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*. Upper Saddle River, NJ: Prentice Hall.

Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology and Design*. Upper Saddle River, NJ: Prentice Hall.

- Kavantzaz, N., Burdett, D. and Ritzinger, G. 2004. 'Web Services Choreography Description Language Version 1.0'. <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>.
- Lovelock, C., Vandermerwe, S. and Lewis, B. (1996). *Services Marketing*. Englewood Cliffs, NJ: Prentice Hall.
- Newcomer, E. and Lomow, G. (2005). *Understanding SOA with Web Services*. Boston: Addison-Wesley.
- Owl_Services_Coalition. 2003. 'OWL-S: Semantic Markup for Web Services'. <http://www.daml.org/services/owl-s/1.0/owl-s.pdf>.
- Pautasso, C., Zimmermann, O. and Leymann, F. (2008). 'RESTful Web Services vs "Big" Web Services: Making the Right Architectural Decision'. *Proc. WWW 2008*, Beijing, China: 805–14.
- Richardson, L. and Ruby, S. (2007). *RESTful Web Services*. Sebastopol, Calif.: O'Reilly Media Inc.
- Turner, M., Budgen, D. and Brereton, P. (2003). 'Turning Software into a Service'. *IEEE Computer*, 36 (10), 38–45.
- White, S. A. (2004a). 'An Introduction to BPMN'. <http://www.bpmn.org/Documents/Introduction%20to%20BPMN>.
- White, S. A. (2004b). 'Process Modelling Notations and Workflow Patterns'. In *Workflow Handbook 2004*. Fischer, L. (ed.). Lighthouse Point, Fla.: Future Strategies Inc. 265–294.
- White, S. A. and Miers, D. (2008). *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Lighthouse Point, Fla.: Future Strategies Inc.

嵌入式软件

目标

本章的目标是介绍嵌入式实时系统的重要特征和实时软件工程，读完本章，你将了解以下内容：

- 理解嵌入式软件的概念，嵌入式软件用于必须对那些来自其环境的外部事件做出反应的控制系统；
- 介绍实时系统的设计过程，软件系统组织成一组协作的过程；
- 理解常用于嵌入式实时系统设计的三大体系结构模式；
- 理解实时操作系统的构成以及它们在嵌入式实时系统中的作用。

从简单的家用电器到整个制造工厂，采用计算机控制的系统非常广泛。这些计算机直接与硬件装置交互。它们需要对硬件产生的事件做出响应并发出相应的控制信号。这些信号引发动作，例如，开始一个电话呼叫、在屏幕上角色的移动、阀门的打开、系统状况的显示。这些系统中的软件嵌入到硬件中，通常是只读存储器里，并且总是能实时地响应系统环境中的事件。在实际应用中，软件系统响应外部事件有一个时限。如果超过了这个时限，整个硬件软件系统将不能正确地运行。

现在由于几乎每种电器设备都包括软件，嵌入式软件变得非常重要。与其他类型的软件系统相比，嵌入式软件系统更多。如果环视屋内，你可能有三四台个人电脑。但是你可能有 20 或 30 个嵌入式系统，如电话、炉灶、微波炉等系统。

实时响应是嵌入式系统和其他软件系统之间的一个最为重要的不同点。这里所说的其他系统，如信息系统、基于 Web 的系统或者个人软件系统，它们的主要目的是数据处理。对于非实时系统，它们的正确性可以通过指定系统输入如何映射到由系统产生的对应的输出来定义。对一个输入的响应，系统应该产生对应的一个输出，通常应该存储一些数据。例如，在病人信息系统中，如果你选择一条创建命令，那么正确的系统响应是在数据库中创建一条新的病人记录，并证实这已经实现。在合理的限度内，它的执行与时间无关。

然而，在实时系统中，准确性既依赖于对输入的响应，又依赖于产生该响应的的时间。如果系统要花费太长时间去响应的话，那么所要的响应可能是无效的。例如，如果嵌入式软件控制的汽车刹车系统执行太慢，那么由于车不能及时停下来，就会发生事故。

因此，在实时软件系统的定义中，时间是固有要素：

实时系统是一个软件系统，系统的功能正确与否取决于系统产生的结果以及产生这个结果的时间。“软”实时系统是这样一个系统，如果结果没有在规定时间内产生，则它的操作被视为退化的操作，“硬”实时系统是这样一个系统，如果结果没有在规定时间内产生，则它的操作被视为系统失败。

对于所有嵌入式系统来说，及时响应是一个重要的因素，但是，在某些情况下，却不需要非常快速的响应。例如，在前面多章中使用过的胰岛素泵系统就是一个嵌入式系统。然而，在它需要对血糖值做检测的周期性间隔当中，不需要对外部事件进行十分迅速的响应。野外气象站软件也是一个嵌入式系统，但是它也不要求对外部事件进行迅速的响应。

除了包含实时响应的需求以外，嵌入式系统和其他类型的软件系统之间还有其他的重要不同点：

1. 嵌入式系统通常持续运行而不终止。当硬件开启时，系统开始执行，且必须执行直到硬件被关闭。这就意味着可靠的软件工程技术，正如第 13 章中介绍的，必须用来确保连续运行。实时系统可能包括更新机制来支持动态配置，这样系统处于服务期间也能得到更新。

2. 和系统环境的交互是不可控制和不可预测的。在交互系统中，交互的节奏是受系统控制的。通过限制用户选择，待处理的事件是可以提前了解的。相反，实时嵌入系统必须能对任何时候的意外事件做出反应。这导致了对实时系统的设计是基于并发性的，多个进程并行执行。

3. 存在影响系统设计的物理限制。这样的例子包括系统可利用的电源限制和硬件占据的物理空间的限制。这些限制可能会产生对嵌入式软件的需求，例如，节约能量以延长电池供电时间的需求。尺寸和重量的限制意味着软件要承担一些硬件功能，因为系统需要限制使用的芯片数量。

4. 直接硬件交互是必要的。在交互式系统和信息系统中，有一个软件层（设备驱动）对操作系统隐藏了硬件。这是可能的，因为你只能将几种类型的设备连接到系统，例如键盘、鼠标、显示器等。相反，嵌入式系统要和各种各样的硬件设备交互，而这些设备没有单独的设备驱动。

5. 安全性和可靠性在系统设计中占据主要的地位。许多嵌入式系统控制的设备如果失败了会带来高昂的人力或经济成本。因此，可依赖性是非常重要的，并且系统设计必须确保设备一直都保持着安全要求极高的行为。通常采用保守的方法设计，即使用调试过或测试过的成熟的技术，而不是使用可能会导致失败模式的新技术。

嵌入式系统可以看做是反应式系统，即它们必须以环境所要求的特定的速度对环境中的事件做出反应（Berry, 1989; Lee, 2002）。响应时间通常是由物理学定律所控制的而非为人的便捷所选择的。这一点与其他软件系统很不一样，其他类型软件系统控制着交互速度。例如，作者用于写书的字处理器能够检查拼写和语法，对处理所用的时间没有限制。

20.1 嵌入式系统设计

嵌入式系统的设计过程是一个系统工程的过程，在该过程中软件设计者要深入考虑系统硬件的设计和性能。部分系统设计过程是要决定哪些系统能力要用软件实现，哪些要用硬件实现。对于很多嵌入在消费类产品（如手机）中的实时系统来说，硬件的成本和电能消耗是一个非常重要的问题。需要使用专用的处理器来支持嵌入式系统，而对于某些系统，可能还需要设计和建造专用的硬件。

这意味着自上而下的软件设计过程，即设计开始于抽象模型，通过一系列阶段对抽象模型分解和开发。这一过程对于绝大多数实时系统来说是不现实的。对于硬件、支持软件以及系统时序等方面的一些下层决策，需要在过程的早期考虑。这些限制了系统设计者的灵活性，可能意味着一些额外的软件功能，比如电池和电源管理必须包括在系统内。

考虑到嵌入式系统是对它们的环境中的事件做出反应的反应式系统，对嵌入式实时软件设计的一般方法是基于激励-响应模型。激励是发生在软件系统环境中引起系统以某种方式响应的事件；响应是由软件发送到它的环境的一种信号或消息。

我们可以通过列出由系统接收的激励和相关联的响应，以及响应必须产生的时间，来定义一个实时系统的行为。例如，图 20-1 给出了防盗报警系统的可能的激励和系统响应。20.2.1 节给出了更多关于该系统的信息。

激 励	响 应
单个传感器结果为正	启动警报；打开此传感器附近的照明灯光
两个或更多的传感器结果为正	启动警报；打开此传感器附近的照明灯光。报告警察有可疑闯入的位置
电压降 10% ~ 20%	切换到备份电池供电；运行电源供电测试
电压降幅超过 20%	切换到备份电池供电；启动警报；报告警察；运行电源供电测试
电源供电失败	报告服务技术人员
传感器失败	报告服务技术人员
控制台紧急按钮被按下	启动警报；打开控制台附近照明灯光；呼叫警察
清除警报	关掉所有开启的警报；关掉所有打开的照明灯光

图 20-1 防盗报警系统的激励和响应

激励有两类：

- 1. 周期性的激励 在每个可预测的时间间隔内发生。举例来说，系统可能每隔 50 ms 检查一次传感器，根据传感器的值（激励）采取响应行动。
- 2. 非周期性的激励 这种激励是不规则地发生的，也是不可预测的。通常使用计算机的中断机制发送信号。这种激励的例子是一个中断，它指示输入/输出传输完成且数据已经在缓冲器中了。

如图 20-2 所示，激励来自于系统环境中的传感器，响应被发送到执行器（actuator）。实时系统的一般设计准则是对于每一类传感器和执行器有不同的进程（见图 20-3）。这些执行器控制设备，例如泵，由此对系统环境做出改变。执行器自身也可以产生激励。来自于执行器的激励通常提示某些问题的发生必须由系统处理。

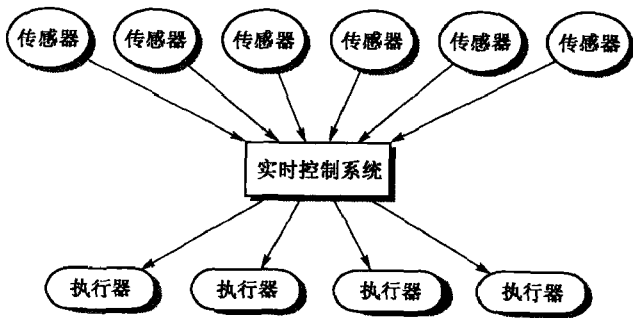


图 20-2 嵌入式实时系统的一般模型

对每一类传感器，有一个传感器管理进程处理来自于传感器的数据采集。数据处理进程计算收到激励所需要的响应。执行器控制进程是与每个执行器相关联的并且管理执行器的操作。这个模型能从传感器迅速采集数据（在被下个输入覆盖之前），并允许之后对数据的加工以及相关执行机构的响应执行。

实时系统需要响应不同时间里发生的激励，因此，它的体系结构的组织需要保证只要接收到激励，立即将相应的控制传到适当的处理单元中去。这在顺序程序中是不现实的，因

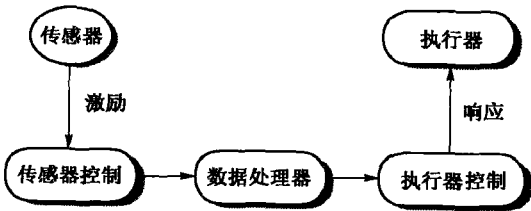


图 20-3 传感器和执行器进程

此, 正常情况下, 实时软件系统都被设计成一组并发协作的进程。为了支持这些进程的管理, 实时系统所处的执行平台可能包括实时操作系统 (将在 20.4 节讨论)。实时操作系统所提供的功能通过对所用的实时编程语言的实时支持系统访问。

并没有标准的嵌入式系统设计过程。使用不同的过程取决于系统的类型、可利用硬件以及开发系统的机构。实时软件设计过程可能包含以下几步:

1. **平台选择** 此活动中, 为系统选择一个执行平台 (即要使用的硬件和实时操作系统)。影响这些选择的因素包括系统的时序限制、可用电源的限制、开发团队的经验、交付系统的价格目标。

2. **激励/响应识别** 包括识别系统必须处理的激励和对每种激励相应的一个或多个响应。

3. **时序分析** 对于每种激励和相应的响应, 找到应用于激励和响应处理的时序约束。这些用于在系统中为进程建立时限。

4. **进程设计** 在这个阶段, 将激励和响应处理聚集到几个并发的进程中。设计进程体系结构的好的起点是如 20.2 节所描述的体系结构模式。然后, 是对进程体系结构的优化以反映出你需要实现的特殊需求。

5. **算法设计** 对于每个激励和响应设计算法执行所需要的运算。算法设计需要在设计过程中相对早些进行, 给出要求处理的数量的提示和完成处理所需要的时间的提示。这对于计算密集型任务 (如信号处理) 是相当重要的。

6. **数据设计** 定义进程交换的信息以及协调信息交换的事件, 并且设计相应的数据结构管理信息交换。几个并发进程可以共享这些数据结构。

7. **进程调度** 设计调度系统确保进程及时开始以满足它们的时限要求。

在实时软件系统设计过程中这些活动的顺序依赖于所要开发的系统的类型、它的过程和它的平台需求。在某些情况下, 我们可能能够遵循一种相对抽象的方法, 即从激励和相关联的处理开始, 并在稍后再决定硬件和执行平台。在其他情形, 对硬件和操作系统的选择是在软件设计开始之前进行的, 在这样一种情况下, 就必须在软件设计中充分考虑到硬件能力所带来的约束。

实时系统中的进程一定是需要协调的也需要共享信息。进程协调机制要确保在共享资源时相互排斥。当一个进程正在修改某个共享资源的时候, 其他进程就不能改变该资源。确保相互排斥的机制包括: 信号灯机制 (Dijkstra, 1968), 监视器机制 (Hoare, 1974), 关键区域机制 (Brinch-Hansen, 1973)。在操作系统的教材中有关于这些机制的详细描述 (Silberschatz 等, 2008; Tanenbaum, 2007)。

当设计进程间信息交换时, 要考虑这些进程以不同的速度在运行这一事实。一个进程产生信息, 另一个进程消费那个信息。如果生产者比消费者运行得快, 在消费者进程读原来的信息之前, 新的信息可能重写先前读入的信息项。如果消费者进程比生产者进程运行快, 同样的信息可能被读两遍。

为了解决这个问题, 应该使用共享缓冲区并且用互斥机制控制对缓冲区访问的方法实现信息交换。这就意味着在信息被读之前不可能被重写并且信息不能被读两次。图 20-4 展示了共享存储区的概念。这通常作为循环队列实现, 这样生产者和消费者进程之间的速度不匹配就会得到调节, 不至于耽误进程执行。

生产者进程一直在缓冲区队列尾部位置存入数据 (在图 20-4 中以 v_{10} 表示)。消费者进程一直从队列头部检索信息 (在图 20-4 中以 v_1 表示)。消费者进程检索信息之后, 列表的头部被调整到下一项 (v_2) 处。当生产者进程增加信息后列表尾部调整到列表的下一空位处。

很明显,确保生产者和消费者进程不同时访问同一项(即,当 $Head = Tail$ 时)是非常重要的。必须确保生产者进程不能向满缓冲区添加项,消费者进程不能从空缓冲区取出项。为了实现这一点,将循环缓冲区实现为一个进程,采用 Get 和 Put 操作访问缓冲区。生产者进程调用 Put 操作,消费者进程调用 Get 进程。同步原语,如信号量或者关键区域,被用于确保 Get 和 Put 操作是同步的,这样它们不会在同一时间访问同一位置。如果缓冲区满了,Put 进程就等待一直等到槽变空;如果缓冲区空,Get 进程一直等待直到产生了一个条目。

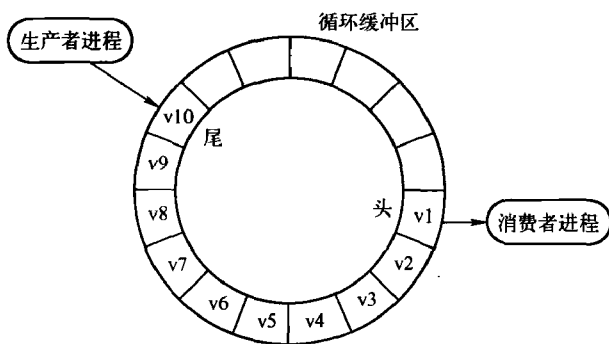


图 20-4 生产者/消费者进程共享循环缓冲区

一旦为系统选择了执行平台,设计了进程的体系结构并确定了调度策略,接下来就要检查系统是否能满足它们的时序需求。我们可以根据组件的时序行为知识通过静态分析来做此项工作,也可以通过仿真来完成。这样的分析能暴露出系统不能很好地运行的情形。进程的体系结构、调度策略、执行平台或者所有这些都必须重新设计以便提高系统的性能。

时序约束或者是其他需求有时意味着我们最好是用硬件而不是软件来实现某些系统功能,比如信号处理。最新的硬件组件,例如 FPGA,它是柔性的,可以适应不同的功能。硬件组件较之软件组件来说能带来更好的性能。可以找到系统处理瓶颈并采用硬件来更换,这样能避免昂贵的软件优化。

20.1.1 实时系统建模

实时系统需要响应的事件通常引起系统从一个状态转移到另一个状态。因为这个原因,如第5章描述的,状态模型通常用来描述实时系统。系统的状态模型假设为,系统在任一时刻一定处于多个可能状态的某个状态中。当接收到一个激励时,系统可能就会产生一次状态转换。举例来说,当操作人员给出命令(激励)的时候,控制阀门的系统可能从“阀门开启”状态转到“阀门关闭”状态。

系统的状态模型是一个很好的语言无关的表达实时系统设计的方式,因此它是实时系统设计方法的一个基本组成部分(Gomaa, 1993)。UML 使用状态图支持状态模型的开发(Harel, 1987; Harel, 1988)。状态图是形式化的状态机模型,支持层次化的状态,这样多个状态组可以当成一个单个实体。Douglass 讨论了在实时系统开发当中的 UML 的使用(Douglass, 1999)。如第5章描述的,状态机模型用于模型驱动的工程来定义系统的操作。这些模型能自动转换成可执行程序。

第5章已经用简单的微波炉模型来说明了这个系统建模方法。图20-5是另一个状态机模型的例子,它给出嵌入在汽油(或燃气)泵中的油料输送软件系统的操作。圆角矩形代表系统状态,箭头代表激励,它产生从一个状态到另一个状态的转换。在状态机图中所选择的名称是描述性的,相关联的信息指出由系统执行单元所执行的动作或者是显示的信息。注意,这个系统从来不会终止,只是在泵不工作时处于等待状态。

油料传递系统是为无看守运行而设计的。购买者将信用卡插入油泵的读卡机。这引起状态转移到 Reading 状态,此时卡的详细信息被读出并请购买者收回卡。取回信用卡触发一个转移到 Validating 状态的转换,此时验证卡是否有效。如果卡是有效的,系统就初始化泵,当油料软管

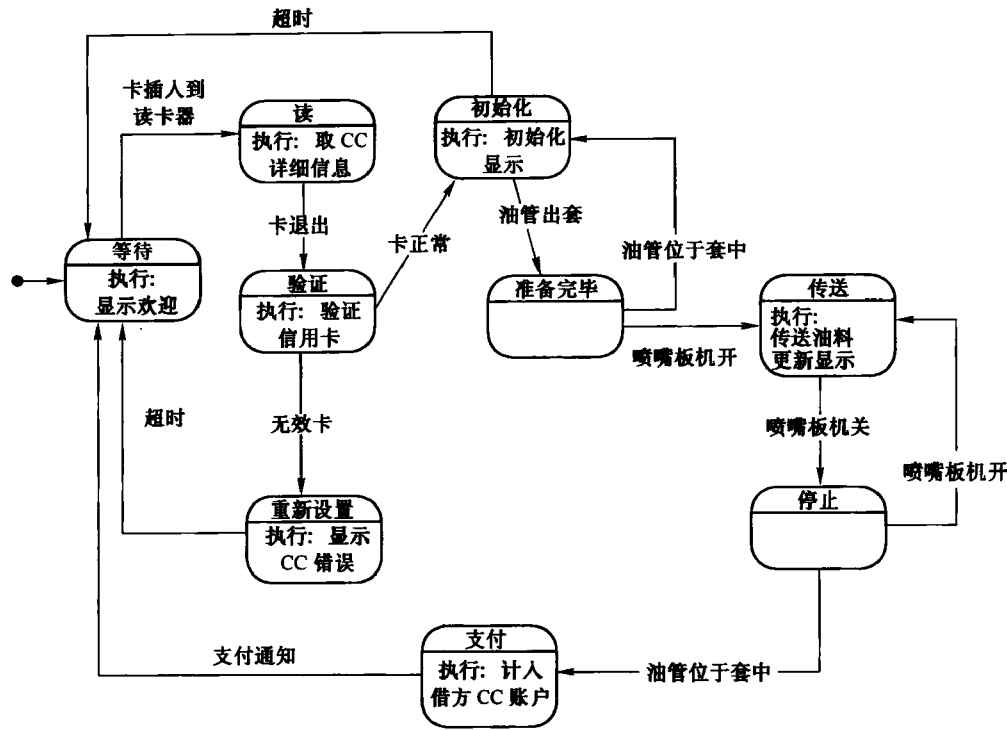


图 20-5 汽油（燃气）泵状态机模型

从皮套中移出，系统转换到 Delivering 状态，准备输送油料。激活位于管口的扳机，油料就会泵出。当扳机释放时油料输送就停止。（为简化起见，忽略了为终止油料溢出而设计的压力开关。）在油料输送完成之后，购买者将油管送回皮套，系统转移到 Paying 状态，此时油料所用金额从用户账户划转。付款之后，泵软件转换到 Waiting 状态。

20.1.2 实时编程

用于实时系统开发的程序语言必须包括能访问系统硬件的能力，而且它要能够预测语言中的特别操作的时序。硬实时系统现在还仍然使用汇编语言来编程，这是因为，只有使用汇编语言才能满足苛刻的时限要求。系统级语言，比如 C 语言，它能生成高效的代码，所以也是广泛使用的。

使用像 C 语言这样的低级系统编程语言的优势是，它能够开发出高效的程序。然而，这些语言不包含支持并发或资源共享管理的结构。并行性和资源管理是通过调用实时操作系统提供的原语实现的，例如互斥信号量。而这种对实时操作系统的调用是不能被编译器检查的，所以，很容易产生编程错误。而且，这种程序通常也是很难理解的，因为语言中并不包含实时特征。在理解程序的同时，读者还要知道如何使用系统调用提供对实时性的支持。

因为实时系统必须满足它们的时序约束，对于硬实时系统就不能使用面向对象的开发。面向对象开发会通过对象中所定义的操作隐藏数据表示和访问的属性值。这意味着在面向对象系统中有非常大的开销，因为需要额外的代码协调属性的访问和处理对操作的调用。由此在性能上的损失使之不可能满足实时的时限要求。

已经有了为嵌入式系统开发所设计的 Java 版本（Dibble, 2008）。不同的公司，如 IBM 和 Sun，都有其不同的 Java 实现。该语言包含了已修改的线程机制，允许所指定的线程不被语言垃

圾收集机制所中断。也包括异步事件处理机制和时序描述。然而，在写本书之际，该语言大多用于有强大处理器和存储能力的平台（例如手机），而不是仅有有限资源的简单嵌入式系统。后者通常仍是用 C 语言实现的。



实时 Java

Java 编程语言已经通过多种方式改编为适应实时系统开发的语言。这种修改包括异步通信；加入时间，包括绝对时间和相对时间；新的不允许被垃圾回收中断的线程模型；一种新的能避免由于垃圾回收所带来的不可预知的延迟的内存管理模型。

[http://www. SoftwareEngineering-9. com/Web/RTS/Java. html](http://www.SoftwareEngineering-9.com/Web/RTS/Java.html)

20.2 体系结构模式

第6章介绍的体系结构模式是对好的设计实践的抽象的格式化的描述。它们封装了系统体系结构的组成、什么时候应该使用这些体系结构，以及它们的优势与劣势。然而，我们不应该把体系结构模式作为一种通用的设计来实现。而是应该通过模式来理解体系结构，并作为创建自己的专门的体系结构设计的起点。

正如所期待的那样，嵌入式和交互式软件的不同意味着，对于嵌入式系统我们需要使用不同的体系结构模式，而不是在第6章谈论的那种体系结构模式。嵌入式系统模式是面向过程的，而不是面向对象或者面向组件的。在这一节将讨论3种经常使用的实时的体系结构模式。

1. **观察和反应** 该模式用于当一组传感器周期性地监控和显示的时候。当传感器显示已经发生了某个事件时（如电话来电），作为反应系统会启动一个进程来处理此事件。

2. **环境控制** 该模式用于某些系统，这些系统包括能提供环境信息的传感器和改变环境的执行器。根据传感器检测到的环境改变，控制信号被发送到系统的执行器。

3. **处理管道** 该模式用于在数据被处理之前需要从一种表示变换到另一种表示的时候。变换实现为一系列可并行执行的处理步骤。由于单独的核或处理器可以执行每个变换，因此该模式允许非常快速的数据处理。

当然这些模式可以结合使用，并且通常你可以看到在一个系统中存在多种模式。例如，当使用环境控制模式时，非常普遍的是对于待监控的执行器使用观察反应模式。在执行器失效的事件中，系统可以通过显示一条警告信息、关闭执行器和切换到备份系统等对此事件做出反应。

此处讨论的模式是描述嵌入式系统的总体结构的体系结构模式。Douglass（2002）描述了用于帮助编程者做更详细设计决策的更低层的实时系统设计模式。这些模式是关于执行控制、通信、资源分配和安全性、可靠性等方面的。

这些体系结构模式是嵌入式系统设计的起点；然而它们并不是设计模板。如果当成模板使用，就会设计出低效的过程体系结构。因此，需要优化过程体系结构，从而确保系统没有太多的过程。我们也应该确保系统中的过程、传感器和执行器之间有一个清晰的联系。

20.2.1 观察和反应

监控系统是嵌入式实时系统的一个重要类别。监控系统通过一组传感器来检查它的环境，并且通常以某种方式显示环境的状态。这可以在内置的屏幕上显示，或者是在特定仪器显示器上的显示或者是远程显示。如果系统检测到异常事件或传感器状态，监控系统就会采取某些

行动。通常会产生一个警报提醒操作员注意所发生的事件。有时系统可以启动一些其他的预防动作，例如关闭系统以阻止其受到损害。

观察和反应模式（见图 20-6 和图 20-7）是常用于监控系统的模式。观测传感器的值，当发现特殊的值时，系统以某种方式做出反应。监控系统由多个观察和反应模式的实例组成，系统中的每种传感器都有一个模式的实例。依据系统的需求，我们可以通过合并进程来优化设计（例如，可以用单个显示进程显示来自所有不同类型的传感器的信息）。

名 字	观察和反应
描述	对一组相同类型的传感器的输入值进行收集和分析。这些值以某种方式显示出来。如果传感器的值表示有某个异常状况发生，那么就会启动一个行动来提醒操作员注意，在某些情况下，还会采取某种行动来响应这个异常值
激励	连接到系统的传感器的值
响应	到显示器的输出，报警器触发，到反应系统的信号
进程	观察者（Observer），分析（Analysis），显示（Display），警报（Alarm），反应器（Reactor）
使用场合	监控系统，警报系统

图 20-6 观察和反应模式

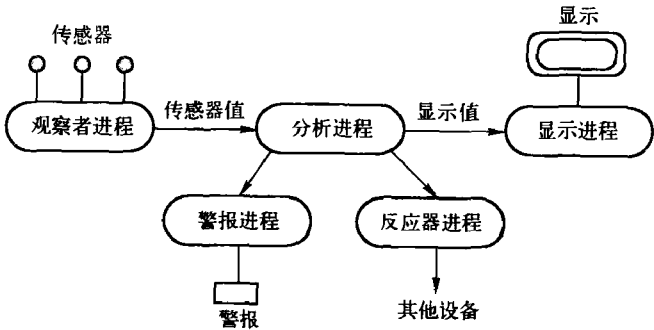


图 20-7 观察和反应的进程结构

这里使用该模式的例子，考虑一个要安装在大厦中的防盗报警系统的设计：

准备实现一个软件系统，该软件系统是安装在商务大厦中的防盗报警系统的一部分。它使用多种不同类型的传感器。这些包括每个房间中的运动传感器，检查走廊门开启的门传感器，安装在一楼的能检测到窗户被破坏的窗传感器。

当传感器检测到入侵者存在时，系统自动呼叫当地警察局，使用声音合成器报告报警的位置。系统打开活动传感器周围的房间内的灯，并发出声音警报。传感器系统使用主电源供电但也配备一个备用电池。使用一个单独的电力电路监视器监视电路的电压从而发现掉电现象。在电压降低到某个水平时触发报警系统。

图 20-8 显示了报警系统的进程结构。图中，箭头表示从一个进程发送到另一个进程的信号。这个系统是一个没有严格时序要求的“软”实时系统。传感器不需要检测高速事件，因此相对来说它们只是偶尔使用。在 20.3 节介绍该系统的时序需求。

在图 20-1 中已经介绍了报警系统的激励和响应。这些是系统设计的起点。在设计中使用了观察和反应模式。存在与每种传感器相关联的观察者进程和与每种反应类型相关联的反应者进程。还有一个单个分析进程检查来自于所有传感器的数据。模式中的所有显示进程被合并为一个显示进程。

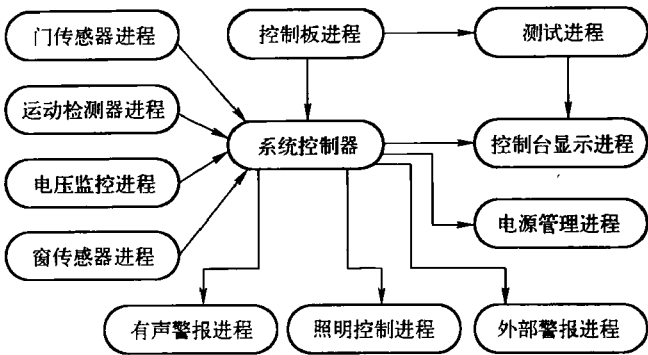


图 20-8 防盗报警系统的进程结构

20.2.2 环境控制

嵌入式软件使用的最为广泛的可能是在控制系统中。在这些系统中，软件基于来自设备所处的环境的激励来控制设备的运行。例如，汽车防滑刹车系统监控汽车的轮子和刹车系统（系统的环境）。它观察给以制动压力时轮子打滑的信号。如果是这种情况，系统调整制动压力，阻止车轮抱死，从而降低侧滑的可能性。

控制系统会使用环境控制模式，即包含传感器进程和执行器进程的通用控制模式。图 20-9 描述了环境控制模式，它的进程体系结构如图 20-10 所示。该模式的一个变种是省略了显示进程的体系结构。在不要求用户干预或控制频率很高以至于显示无意义的情况下，使用此变种模式。

名 称	环 境 控 制
描述	系统分析来自一组采集系统环境数据的传感器的信息。进一步的信息也可能是来自连接到系统的执行器的状态信息。基于传感器和执行器的数据，传送控制信号到执行器，进而引起系统环境的改变。关于传感器的值和执行器的状态的信息会显示出来
激励	连到系统上的传感器的值以及系统执行器的状态
响应	给执行器的控制信号，显示信息
进程	监控器（Monitor），控制（Control），显示（Display），执行器驱动（Actuator Driver），执行器监视器（Actuator Monitor）
使用场合	控制系统

图 20-9 环境控制模式

这个模式可以作为控制系统设计的基础。基本的做法是针对每一个控制执行器（或执行器类型）给出一个环境控制模式实例。进一步的优化做法是减少进程的数量。例如，可以合并执行器监控进程和执行器控制进程，或者对几个执行器使用一个监控和控制进程。选择的优化取决于时序需求。可能监控传感器比发送控制信号更频繁，在此情形下，结合控制和监控进程就是不可行的了。在执行器控制和执行器监控进程之间也会存在直接

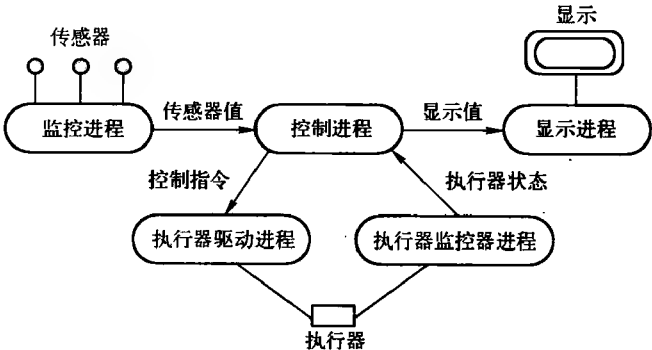


图 20-10 环境控制的进程结构

反馈。这就允许我们通过执行器控制进程给出细粒度的控制决策。

图 20-11 示意了该模式是如何使用的。这是一个汽车刹车系统控制器的例子。设计的起始点是针对系统中的每种执行器类型给出一个模式实例。在这儿有 4 个执行器，每一个控制一个车轮上的制动。把单个传感器进程结合到一个监控所有车轮的车轮监控进程，该监控进程监视各个车轮上的传感器。传感器进程监控每一个车轮状态，判断车轮是转动还是抱死。一个单独的进程监视驾驶员踩刹车踏板的压力。

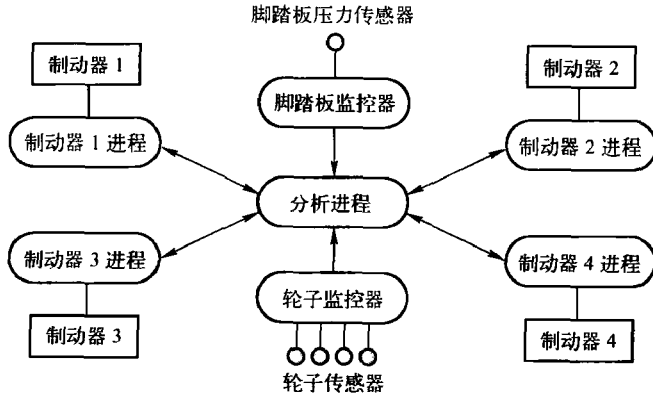


图 20-11 防滑制动系统的控制系统体系结构

系统包括防滑特征，在刹车时如果传感器提示车轮抱死则触发此特征，这意味着在路面和轮胎之间存在着不完全摩擦力；换句话说，车正在滑动。如果轮子被抱死，司机就不能控制轮子。为了对抗这一点，系统给此轮子的制动器发送一个快系列的开关信号，允许轮子转动并且接受控制。

20.2.3 处理管道

许多实时系统主要是从系统环境采集数据，然后将数据从原始数据表示转换成系统便于分析和处理的数字表示。系统也可以将数字数据转换成模拟数据，然后发送给它的环境。例如，软件收音机接受表示无线电传播的数值数据输入包，并且将这些转换成人们能听到的声音信号。

这些系统中，数据处理必须得到快速执行。否则，输入数据可能丢失并且输出符号可能由于基本信息的丢失而被破坏。管道处理模式使得这种快速处理成为可能。它通过把所需要的数据处理分解成一系列独立变换，且每个变换可以由一个独立的进程来执行。这对于使用多重处理器和多核处理器的系统是一个非常高效的体系结构。管道中每一个进程被分配到单独的处理器或者核上运行，因此处理步骤能并行执行。

图 20-12 是对数据管道模式的简洁描述，图 20-13 显示了这个模式的进程体系结构。注意，进程能够产生和消费信息。如同 20.1 节讨论的，它们与同步缓冲区连接。这就允许生产者消费者进程以不同的速度工作而无数据损失。

高速数据采集系统是使用管道处理模式的系统的一个例子。数据采集系统从传感器采集数据，然后进行后续的处理和分析。这些系统可以用于这样的情形，即传感器从系统环境中采集大量数据但不可能或者没有必要进行实时处理，而是采集并存储以便今后用于分析。数据采集系统通常用在科学实验和过程控制系统中。过程控制系统中的物理过程，例如化学反应，是非常快速的。在这些系统中，传感器快速生成数据，数据采集系统需要确保在传感器值改变之前将传感器的读数采集出来。

名 称	处 理 管 道
描述	处理管道使得数据以序列形式从管道的一端移动到另一端。这些处理通常关联到一些同步缓存上，允许生产者和消费者进程以不同的速度运行。标志管道完成的是显示或数据存储，或者是管道连接到一个执行器
激励	输入来自环境或者其他进程的值
响应	输出值给环境或者给一个共享缓存
进程	生产者（Producer），缓存（Buffer），消费者（Consumer）
使用场合	数据获取系统，多媒体系统

图 20-12 管道处理模式

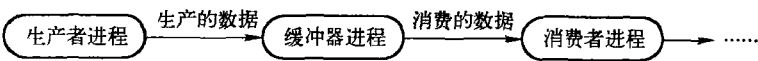


图 20-13 处理管道模式的进程结构

图 20-14 是一个作为核反应堆中控制软件一部分的数据采集系统的简化模型。这是一个从监视反应堆中中子通量（即中子密度）的传感器采集数据的系统。传感器数据放置在一个缓冲区中，之后数据从该缓冲区提取和处理。平均通量水平显示在控制台显示器上，并且存储起来为以后处理用。

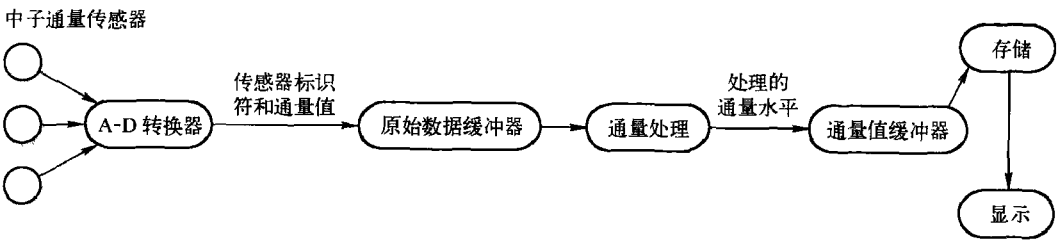


图 20-14 中子通量数据获取

20.3 时序分析

正如本章的引言中所提到的，实时系统的正确性并不仅仅依赖于输出的正确性，而且还取决于产生输出的时间。这就意味着在嵌入式实时软件开发过程中的一个重要方面是时序分析。在这个分析中，计算系统中每个进程的执行频率来确保所有的输入得到处理并且所有的系统响应及时产生。时序分析的结果决定每个进程应该执行的频率，以及实时操作系统应该如何调度这些进程。

当系统要混合处理周期性的和非周期性的激励与响应时，实时系统的时序分析是相当困难的。由于非周期性激励是不可预测的，因此必须假设这些激励发生并因而在任何特定时间要求服务的可能性。这些假设可能是不正确的，系统性能在系统交付后可能不够好。Cooling（2003）讨论了考虑非周期性事件的实时系统性能分析的技术。

然而随着计算机处理速度越来越快，在许多系统中，使用周期性激励的设计已成为可能。由于处理延迟通常给系统带来某些损失，当处理器很慢的时候，就需要采用非周期性激励以确保重要事件在它们的时限之前得到处理。例如，在嵌入式系统中电源供应失败意味着系统在很短的时间内（50 毫秒）以一种控制方式关闭相关设备。这可以实现为“电源失败”中断。然而，这也可以用一个周期性进程来实现，该进程以非常快速的方式运行来检查电源。只要进程调用

之间的时间很短，在掉电引起损害之前，仍然有时间执行系统的控制性关闭。正因如此，这里只研究周期性进程的时序问题。

当分析嵌入式实时系统的时序需求和设计系统满足这些需求的时候，要考虑3个主要因素：

1. **时限** 这是一个时间，在此之前激励必须得到处理并且响应必须由系统产生。在系统不能满足时限情况下，如果系统是硬实时系统，那么系统就失败了；如果系统是软实时系统，那么导致系统服务退化。

2. **频率** 这是每秒系统执行进程的次数，以确保你相信系统能满足时限的要求。

3. **执行时间** 处理激励并且产生响应的的时间。通常，需要考虑两个执行时间，即进程平均执行时间和进程最坏情况下执行时间。由于代码的条件性执行以及延迟以等待其他进程等原因，执行时间并不一直是相同的。在硬实时系统中，要做出基于最坏情况下执行时间的假设来确保时限不会被突破。在软实时系统中，你能够基于平均执行时间得到你的计算结果。

接着用供电失败的例子，让我们假设，在发生失败事件后，电压需要50毫秒的时间降到一个能使系统受到损坏的水平。因此，设备关闭进程必须在断电事件后的50毫秒以内开始执行。在这种情况下，由于设备中的物理变数，设置40毫秒或更短的时限是谨慎的。这意味着对所有可能面对风险的相关设备，必须在40毫秒内发出关闭指示并进行处理。

如果通过监视电压水平检测电源失败，那么必须做多个观察以检测电压正在下降。如果每秒运行进程250次，这就意味着运行一次要4毫秒，且你需要至少两个周期来检测电压下降。因此，需要花费8毫秒来检测系统问题。结果，关闭进程的最坏情况执行时间不应该超过16毫秒以确保满足40毫秒的时限。这个数是通过从时限（40毫秒）中减去进程周期（8毫秒）然后除以2，因为执行两次进程是必要的。

实际上，应该设法得到一个远低于16毫秒的执行时间从而留出安全空隙以防计算出错。事实上，用于检查传感器和检测没有电压大幅降低事件的时间远低于16毫秒。这仅仅是两个电压值的比较。电源监控进程的平均执行时间应该小于1毫秒。

在实时系统中，时序分析的起点是时序需求，时序需求设定了每个所要求的响应的时限。图20-15给出了20.2.1节中介绍的商务大厦防盗报警系统的时序需求。这里对设计进行了简化，来自系统自身检测子程序的激励、来自外部的用于对系统测试的激励，以及在系统误发警报情况下关掉系统的激励都忽略了。这样，系统就只剩两类激励需要处理了：

1. **停电** 当电压下降超过20%的时候，系统检测到断电。必需的响应是将电路切换到备用电源上。由电子装置将主电源切换成备用电源。通过向电子装置发出信号来实现这一切换。

2. **侵入者报警** 这是一个产生于系统某个传感器的激励。对这个激励的响应是计算发出异常信息的传感器的房间号码，向警察局发出一个呼叫，激活语音合成器来报告详细信息，同时还要拉响入侵警报和点亮事发周围的灯光。

如图20-15所示，我们通常应该单独对每一类传感器列出其时序约束，即使是（如这里的情形）它们都相同时也是如此。通过对它们单独处理，就能给未来的改变留下空间，使得对每秒内控制进程所需要执行的次数的计算更容易。

接下来的设计阶段是将系统的功能分配到并发进程。有4类传感器需要周期性地轮询，因此，这些传感器都有一个相关的进程。有电压传感器、门传感器、窗传感器和运动检测器。因为一般来讲和传感器相应的进程执行得非常快，它们只负责核实传感器的状态是否改变（例如从关到开）。假定检查和评估一个传感器状态的执行时间不大于1毫秒是合理的。

为了确保满足时序需求定义的时限，接下来需要决定相关进程的运行频率，以及在进程每一次运行期间有多少传感器需要被检测。在频率和执行时间之间有一个明显的权衡。

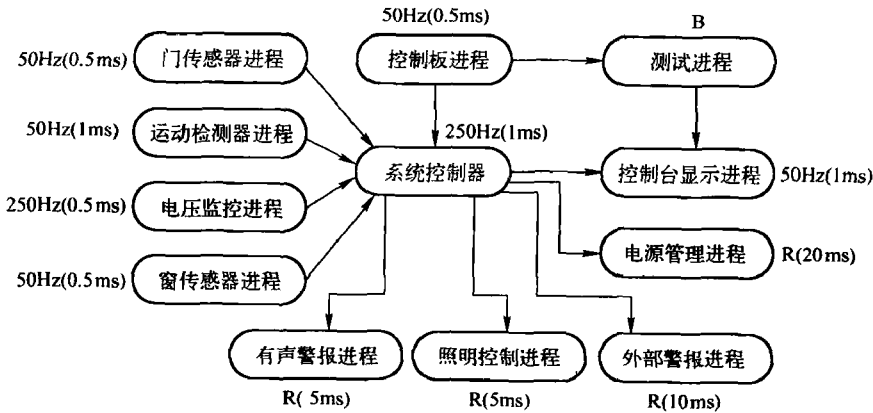
激励/响应	时序需求
电源失败	必须在 50 ms 时间内完成向备用电源的切换
门警报器	每秒需要对每个门警报器轮询两次
窗警报器	每秒需要对每个窗警报器轮询两次
运动检测器	每秒需要对每个运动检测器轮询两次
声音警报器	当有传感器发出警报信号时，声音警报器需要在 0.5 秒内打开
照明开关	当有传感器发出警报信号时，照明开关需要在 0.5 秒内打开
通信	当有传感器发出警报信号时，向警察局的呼叫应该在 2 秒内启动
语音合成	当有传感器发出警报信号时，语音合成的消息应该在 2 秒内可用

图 20-15 防盗报警系统的时序需求

- 1. 如果在每个进程执行过程中检测一个传感器，那么如果有一个指定类型的 N 个传感器，则必须安排进程每秒执行 4N 次以满足在 0.25 秒内检测状态改变的时限。
- 2. 假定在每个进程执行过程中检测 4 个传感器，那么执行时间增加到 4 毫秒，但是每秒仅需要运行进程 N 次来满足时序需求。

在此情形中，由于系统需求定义动作，当两个或多个传感器是正值时，成组地测试传感器是合理的。传感器基于它们在物理位置上的接近程度来分组。如果入侵者进入大楼，那么可能就会有相邻的几个传感器为正值。

完成了时序分析之后，接下来需要利用与执行频率和预期执行时间相关的信息给进程模型做注释（见图 20-16）。此处，周期性进程用它们的频率注释；为响应激励而启动的进程标记为 R；测试进程是后台进程，标记为 B。这意味着仅仅当处理器时间空闲时它才执行。一般来讲，系统进程频率很低的系统设计相对来讲是更简单的。执行次数代表进程所要求的最坏情况下的执行次数。



设计进程的最后一步是设计一个调度系统，以确保进程总是能在其时限内得到调度。只有知道所使用的实时操作系统所支持的调度方法才能做到这一点（Burns 和 Wellings, 2009）。实时操作系统的调度程序为进程分配一个处理器一定的时间。时间能固定，或者依据进程的优先权来变化。

在分配进程优先权时，要考虑每一个进程的时限以便有短时限的进程接收到处理器时间来满足时限。例如，在防盗报警系统中电压监控进程需要调度，这样电压下降能被检测到并且在系统失效前切换到备用电源。由于检查传感器值的进程比起它们的期待执行时间有更宽松的时限，因此比这些进程电压监控进程有更高的优先权。

20.4 实时操作系统

大多数应用系统的执行平台是操作系统。操作系统管理共享资源，提供如文件系统、运行时进程管理等特征。然而在常规操作系统中，大量功能占据了许多空间并使程序的运行变慢。而且，系统的进程管理特征的设计并不允许进行细粒度的进程调度控制。

正因如此，标准的操作系统，如 Linux 和 Windows，通常不用作实时系统的执行平台。非常简单的嵌入式系统不用操作系统，而是直接运行在硬件上。系统自身包括系统启动和关闭、进程和资源管理以及进程调度。然而更常见的是，嵌入式应用建立在实时操作系统（RTOS）上。实时操作系统是一个高效的操作系统，它提供了实时系统所需要的各种特征。Windows/CE、Vx-works 和 RTLinux 都是 RTOS 的实例。

实时操作系统为实时系统管理进程和资源的分配。它启动和停止进程，使得能够处理激励并分配存储器 and 处理器资源给进程。RTOS 的组件（见图 20-17）依赖于所要开发的实时系统的规模和复杂度。除了最简单的之外，所有一般的实时系统总是包括：

- 1. 实时时钟，提供周期性调度进程所需的信息；
- 2. 中断处理器，管理非周期性的对服务的请求；
- 3. 调度单元，该组件负责检查可以执行的进程并选择其中一个去执行；
- 4. 资源管理器，给定一个经过调度准许执行的进程，资源管理器分配相应的内存和处理器资源给进程；
- 5. 分派单元，该组件负责开始一个进程的执行。

对于大型系统的实时操作系统，比如过程控制或电信系统，还会有额外的设施，如磁盘存储管理、检测和报告系统缺陷的缺陷管理设施、支持实时应用的动态配置的管理器等。

进程管理

实时系统必须快速地处理外部事件，在某些情况下，需要在某个时限到来之前处理完这些事件。这意味着事件处理进程必须及时地得到调度执行来检测事件，并需要分配给进程充足的处理器资源保证其在时限之前完成。在 RTOS 中的进程管理器负责选择进程去执行，分配处理器和内存资源，并在处理器上启动和终止进程执行。

进程管理器需要用不同的优先级来管理进程。对于一些激励，比如那些与某些异常事件相关联的激励，对它们的处理在规定的时限之内完成是很重要的。如果有更紧要的进程要求服务时，其他进程可以安全地推迟。因此，RTOS 对于系统进程来说，必须能管理至少两个等级的优

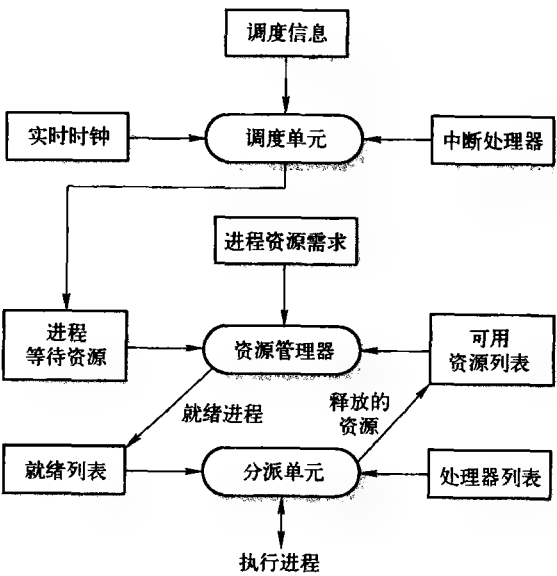


图 20-17 实时操作系统的组件

先权:

1. **中断级** 这是最高的优先级层次。它分派给要求快速响应的进程。实时时钟进程就是这类进程。

2. **时钟级** 这个层次的优先级被分派给周期性的进程。

对于后台进程可能有另外的优先层次（例如自检进程），这种进程没有实时时间限制。当处理器有空闲的时候，这些进程才得以执行。

在每一类优先级中，不同类型的进程会分配到不同的优先顺序。举例来说，可能有多条中断线，为了避免丢失信息，来自快速装置的中断可能会抢占来自慢速装置的一个中断的处理。进程优先次序的分配，可保证所有的进程及时得到服务，这总是需要大量的分析和仿真的。

周期性进程是这样一种进程，它在预先规定的时间间隔上执行数据采集和执行机构控制。在绝大多数实时系统中，将会有多种类型的周期性进程。RTOS 利用应用程序所规定的时序需求安排周期进程的执行，以保证它们都能在时限到来之前完成任务。

操作系统对于周期性进程管理所采取的行动如图 20-18 所示。由调度单元检查周期性进程列表，从中选择一个进程来执行。这个选择取决于就绪进程的进程优先级、进程耗时、期望执行的次数以及截止期限。有时，有两个具有不同的截止期限的进程要在一个时钟节拍中执行。在这种情况下，有一个进程需要被推迟。一般情况下，系统会选择延迟有更长时限的进程。

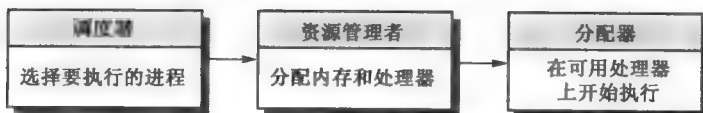


图 20-18 RTOS 启动进程所需要的动作

对非同步事件的响应进程一般都是中断驱动的。计算机的中断机制使得控制被转移给预先安排好的一个内存地址。这个地址包含一条指令来跳转到一个简单而快速的中断服务子程序。该服务子程序首先屏蔽掉进一步的中断，然后找出中断的原因并以一个高优先级来启动一个进程去处理产生此中断的激励。在某些高速数据采集系统中，中断处理器将中断信号所表示的可用数据存储在缓冲器中待后续处理。中断被再次激活并将控制交还给操作系统。

在任何时间里，可能有多个进程会得到执行，每个进程的优先级不同。这时就需要调度单元实现系统调度策略决定这些进程的执行顺序。有两个基本调度策略：

1. **非抢占调度** 一旦一个进程被调度执行，就执行到完成为止，或者是由于某些原因被阻塞为止，例如等待输入。它的问题是当有多个不同优先级的进程时，一个高优先级的进程可能需要等到一个低优先级的进程执行完才能执行。

2. **抢占调度** 当有更高优先级的进程需要服务时，正在执行的进程可能会被终止。这个高优先级进程抢占低优先级进程并被安排到处理器上执行。

在这些策略中，产生了不同的调度算法。包括循环式（round-robin）调度，每个进程轮流执行；速度单调调度，具有最短周期（最高频率）的进程被赋予优先权；最短时限优先调度，队列中进程具有最短时限的优先调度（Burns 和 Wellings, 2009）。

被执行的进程的信息传给资源管理器。资源管理器分配存储器和处理器给进程，在多处理器系统中，需要选择一个处理器来执行该进程。于是，这个进程被放入“就绪列表”中，这是一个包含一组准备执行的进程的列表。当处理器执行完一个进程处于空闲状态时，分派单元被激活。它扫描就绪列表以寻找一个能在这个处理器上执行的进程，然后开始执行这个进程。

要点

- 嵌入式软件系统是对环境中事件做出反应的硬件/软件系统的一部分。软件嵌入在硬件当中。嵌入式系统通常都是实时系统。
- 实时系统是一个必须实时地对事件做出响应的软件系统。它的正确性不仅仅依赖于所产生的结果，而且依赖于产生这些结果所花的时间。
- 实时系统通常实现为一组对激励做出反应以产生响应的互相关联的过程。
- 状态模型是嵌入式实时系统的一种重要的设计表达。系统如何对环境做出反应由事件触发系统状态的改变来描述。
- 在不同类型的嵌入式系统中可以观察到多个标准的模式。它们包括监控系统环境中不利事件的模式、执行器控制模式和数据处理模式。
- 实时系统的设计者一定要进行时序分析，它是由处理和响应激励的时限驱动的。设计者必须做出决策关于系统中的进程应该以什么频度执行以及期待的和最坏的进程执行时间。
- 实时操作系统负责对进程和资源的管理。它总是包括一个调度单元，这个组件负责决定哪个进程应该得到执行。

进一步阅读材料

《Software Engineering for Real-time Systems》从工程的视角而不是从计算机科学的视角撰写，是一本非常好的实时系统工程的实践指导读物。与 Burns 和 Wellings 的书相比，该书对硬件问题的覆盖面更广，所以该书是前者的非常棒的补充读物（J. Cooling, Addison-Wesley, 2003）。

《Real-time Systems and Programming Language: Ada, Real-time Java and C/Real-time POSIX, 4th edition》这是一本全面论述实时系统各个方面的好教材（A. Burns and A. Wellings, Addison-Wesley, 2009）。

《Trends in Embedded Software Engineering》这是一篇关于建议模型驱动开发的论文（如第 5 章中所讨论的），它将成为一个非常重要的嵌入式系统开发方法。该文是嵌入式系统专辑中的一部分，你也可以发现其中还有其他一些论文也是很不错的材料。（IEEE Software, 26 (3), May-June 2009）。<http://dx.doi.org/10.1109/MS.2009.80>。

练习

- 20.1 用例子解释为什么实时系统通常需要使用并发进程来实现。
- 20.2 识别出在家用冰箱或家用洗衣机中的嵌入式系统的可能的激励和响应。
- 20.3 使用基于状态的方法去建模，如在 20.1.1 节中讨论过的，为有线电话中的语音邮件系统的嵌入式软件系统的控制建模。应该在 LED 上显示记录的消息的数量，应该允许用户拨入并接听记录的消息。
- 20.4 解释为什么面向对象软件开发方法可能不适合实时系统的开发。
- 20.5 给出如何使用环境控制模式作为温室温度控制系统的设计基础。温度应该在 10 ~ 30℃ 之间。如果温度低于 10℃，就应该将供暖系统打开；如果温度超过 30℃，窗户就应该自动打开。
- 20.6 设计一个环境监控系统的进程的体系结构，该系统从一组安装在城市周围的空气质量传感器收集数据。5000 个传感器被分成 100 组。每个传感器每秒要被查询 4 次。当某一区域超过 30% 的传感器指示空气质量低于一个可接受的水平时，局部警告灯就被打开。所有传感器将数据返回给中央处理计算机，这台计算机每 15 分钟产生一次该城市的空气质量报告。
- 20.7 一个火车保护系统自动地在达到某个路段速度限制或某一路段当前有红灯显示（该路段不得进入）时对火车实行制动。细节内容如图 20-19 所示。写出火车上控制系统要处理的激励以及相关的响应。

火车保护系统

- 系统需要从轨道旁的发送装置中传出关于路段时速限制的信息，这个发送装置连续不断地广播路段标识和该路段时速限制以及该路段信号状态。需要 50 毫秒广播一次。
- 火车在进入发送装置 10 米范围之内时能接收到信息。
- 火车最大时速为 180 km/h。
- 火车传感器提供火车的当前速度（每 250 毫秒更新一次）和火车制动状态（100 毫秒更新一次）信息。
- 当火车时速超过路段速度限制 5 km/h 以上机车驾驶室的警报就会响起。当超过路段时速限制 10 km/h 以上时制动系统自动开始工作直到将速度降低到路段速度限制之内。制动系统的执行要在 100 毫秒之内开始。
- 当火车进入红灯路段时火车保护系统自动启动制动系统直到火车停下来。制动系统应该在接到红灯信号 100 毫秒内开始工作。
- 系统连续更新驾驶室内的状态显示。

图 20-19 火车保护系统的需求

- 20.8 为练习 20.7 中的系统绘制出一个可能的进程体系结构。
- 20.9 在火车保护系统中，要有一个用来收集从轨道旁发送装置中发送的信息的一个循环过程，这个循环过程需要什么样的执行频度才能确保系统能收集到从传送装置发来的信息？解释如何实现你的答案。
- 20.10 为什么通用目的操作系统，例如 Linux 或 Windows，不适合作为实时系统平台？利用你使用通用目的系统的经验来帮助回答此问题。

参考书目

- Berry, G. (1989). 'Real-time programming: Special-purpose or general-purpose languages'. In *Information Processing*. Ritter, G. (ed.). Amsterdam: Elsevier Science Publishers, 11–17.
- Brinch-Hansen, P. (1973). *Operating System Principles*. Englewood Cliffs, NJ: Prentice-Hall.
- Burns, A. and Wellings, A. (2009). *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Boston: Addison-Wesley.
- Cooling, J. (2003). *Software Engineering for Real-Time Systems*. Harlow, UK: Addison-Wesley.
- Dibble, P. C. (2008). *Real-time Java Platform Programming, 2nd edition*. Charleston, SC: Booksurge Publishing.
- Dijkstra, E. W. (1968). 'Cooperating Sequential Processes'. In *Programming Languages*. Genuys, F. (ed.). London: Academic Press, 43–112.
- Douglass, B. P. (1999). *Real-Time UML: Developing Efficient Objects for Embedded Systems, 2nd edition*. Boston: Addison-Wesley.
- Douglass, B. P. (2002). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston: Addison-Wesley.
- Gomaa, H. (1993). *Software Design Methods for Concurrent and Real-Time Systems*. Reading, Mass.: Addison-Wesley.
- Harel, D. (1987). 'Statecharts: A Visual Formalism for Complex Systems'. *Sci. Comput. Programming*, 8 (3), 231–74.
- Harel, D. (1988). 'On Visual Formalisms'. *Comm. ACM*, 31 (5), 514–30.
- Hoare, C. A. R. (1974). 'Monitors: an operating system structuring concept'. *Comm. ACM*, 21 (8), 666–77.
- Lee, E. A. (2002). 'Embedded Software'. In *Advances in Computers*. Zelkowitz, M. (ed.).

London: Academic Press.

Silberschatz, A., Galvin, P. B. and Gagne, G. (2008). *Operating System Concepts, 8th edition*. New York: John Wiley & Sons.

Tanenbaum, A. S. (2007). *Modern Operating Systems, 3rd edition*. Englewood Cliffs, NJ: Prentice Hall.

面向方面的软件工程

目标

本章的目标是介绍面向方面的软件开发。面向方面的软件开发将关注点分散到独立系统模块。当你阅读完本章，你将了解到以下内容：

- 为什么关注点的分离对软件开发是个很好的指导原则；
- 介绍方面和面向方面软件开发背后的基本思想；
- 理解面向方面的方法怎样在需求工程、软件设计和编程中应用；
- 了解面向方面的系统测试中的困难。

在绝大多数大系统中，需求和程序组件之间的关系是复杂的。单个需求可能是通过很多组件实现的，每个组件可能包含几个需求的要素。这意味着，在实践中，实现对需求的一个变更可能需要了解和改变若干组件。换句话说，一个组件可以提供某些核心功能，但也包含实现多个系统需求的代码。即使看来存在巨大的复用的可能性，复用这样的组件的成本也会是非常昂贵的。复用会涉及修改组件以删除其中与组件的核心功能不相关的额外代码。

面向方面的软件工程（Aspect-Oriented Software Engineering, AOSE），是一种旨在解决这个问题并使程序易于维护和复用的软件开发方法。AOSE 基于一种称为方面的抽象。方面实现那些可能在程序中若干个不同地方用到的系统功能。方面封装功能，这些功能与系统中包含的其他功能交叉和共存，是和其他的抽象（如，对象和方法）一起使用的。一个可执行的面向方面的程序是根据包含在程序源代码中的描述，通过自动组合（或称为编织）对象、方法和方面而创建出来的。

方面的一个最主要的特性是它们包含一个定义，规定应该在程序的什么地方包括它们，同时也包含实现横切关注点的代码。这样你就可以指定应该在一个特定方法调用之前或之后或在对某个属性访问之时包含横切代码。本质上讲，方面通过编织到核心程序中来创建一个新的增强的系统。

面向方面方法的最主要的好处是它支持关注点的分离。如 21.1 节中解释的，较之将不同的关注点堆积在同一个逻辑抽象中，将关注点分离到独立的要素中是一种好的软件设计实践。通过把横切关注点表示为方面，这些关注点就能够被独立地理解、复用和修改，而不用关心代码在何处使用。例如，假设用户认证可以表示为一个要求登录用户名和口令的方面。这能够自动地编织到程序中，只要是要求用户认证的地方，无论哪里都可以使用它。

比如有一个需求，需要在数据库中任何个人详细资料发生改变之前对用户进行身份认证。我们可以通过一个声明在方面中描述这项需求，声明为：应该在每个对个人详细资料更新的方法调用之前将认证代码包含进去。这样，我们就可以扩展认证需求到所有的数据更新。这可以通过修改方面很容易地实现。我们无需在整个系统中搜索来寻找所有发生这些方法调用的地方。因此我们可以减少犯错误的机会和意外地把安全脆弱性带入程序的可能。

面向方面的研究和开发主要集中于面向方面的编程。面向方面的编程语言，例如 AspectJ（Colyer 和 Clement, 2005；Colyer 等, 2005；Kiczales 等, 2001；Laddad 2003a, Laddad 2003b）已经开发出来，它们是对面向对象编程语言的扩展使之包含了方面的特征。一些主要的大型公司

已经开始在它们的软件生产过程中使用面向方面的编程 (Colyer 和 Clement, 2005)。但是, 横切关注点在软件开发的其它阶段同样是存在问题的。研究人员正在研究怎样在系统需求工程和系统设计中利用面向方面技术, 如何测试和检验面向方面的程序。

在此讨论 AOSE, 是因为它的核心内容——分离关注点, 是思考和构建软件系统的重要方法。尽管一些大型系统是用面向方面的方法实现的, 方面的使用仍然不属于主流软件工程范畴。和其他新技术一样, 比起存在的问题和花费, 提倡者更关注其优势。尽管 AOSE 要在软件工程中与其他方法一起常规性使用还需要一定的时间, AOSE 中包含的关注点分离思想是很重要的。关注点分离思想是软件工程中一个好的通用方法。

所以在本章剩下的各节中, 关注一些 AOSE 的概念, 并讨论在软件开发过程的不同阶段使用面向方面方法的优势和劣势。因为目标是帮助读者理解 AOSE 深层的概念, 所以不会具体讨论任何特定的方法和面向方面的编程语言。

21.1 关注点分离

关注点分离是软件设计和实现中的一个重要原则。它意味着你应该这样组织你的软件, 使得程序中的每个元素 (类、方法、过程等) 做且只做一件事。这样你就能够集中注意力在一个元素上而无需考虑程序中的其他元素。你可以通过了解其关注点来了解程序的每个部分, 而无需了解其他元素。当需要进行某些改变时, 变更只局限在少数的元素上。

分离关注点的重要性在计算机科学历史的很早阶段就被人们认识到了。封装一个最小功能的子程序在 20 世纪 50 年代早期就发明了, 随后的程序构造机制, 例如程序和对象类, 也设计用来为实现分离关注点提供更好的机制。但是, 所有这些机制在处理某些类型关注点上都存在问题, 这种关注点横贯其他关注点, 我们称之为横切关注点 (cross-cutting concern)。这种横切关注点不能使用例如对象或函数等构造机制被局部化。于是人们发明了方面的概念以帮助管理这类关注点。

虽然人们已经普遍赞同分离关注点是个很好的软件工程实践, 但清楚地界定关注点的真正内涵是很难的。有时, 它被定义为一个功能概念 (比如, 关注点是系统中的某些功能要素)。同样的, 它也可以被定义得非常宽泛, 如定义它是“程序中任意一块兴趣点或焦点片段”。这两个定义都不是很有用。关注点肯定不仅仅是简单的功能元素, 但更通用的定义又太含糊以至于无实用价值。

在作者看来, 绝大多数定义关注点的尝试是存在问题的, 因为它们试图将关注点与程序相关联。实际上, 就像 Jacobsen 和 Ng (2004) 所论述的, 关注点是系统信息持有者对系统需求和优先权的真实反映。系统性能可以是一个关注点, 因为使用者希望能够得到系统的及时响应; 某些项目信息持有者可能关心的是系统应该包括某特殊功能; 对系统提供技术支持的公司, 可能关注的是系统能够很容易地维护。因此, 关注点可以定义为一个或一组项目信息持有者感兴趣的或者对他们意义重大的事情。

如果把关注点看做是组织需求的一种方式, 我们可以看出为什么将关注点分离实现在不同的程序元素中的方法是个很好的实践。当用关注点来表示一个需求或一组相关需求的时候, 我们就可以很容易在实现这些关注点的程序组件中跟踪需求。如果需求改变了, 程序的必须改变的部分是明显的。

下面是多种不同的信息持有者关注点:

1. 功能性关注点, 它是将包含在系统中的与特殊功能相关的关注点。例如, 在火车控制系统中, 一个特殊的功能性关注点是火车制动。
2. 服务质量关注点, 它是与系统的非功能性行为相关的关注点。这些包括性能、可靠性、

可用性等特征。

3. 政策关注点, 这是与管理系统使用的总的政策相关的关注点。政策关注点包括信息安全关注点、安全关注点以及业务规则相关的关注点。

4. 系统关注点, 这是与系统总体属性相关的关注点, 例如它的可维护性和可配置能力。

5. 机构关注点, 这是与机构的目标和优先级相关的关注点, 例如在预算范围内完成系统构建, 对现存的软件资产的利用, 或者是维持机构声誉。

系统的核心关注点是那些关系到它的主要目标的功能性关注点。因此, 对于一个医院病人信息系统来说, 核心功能性关注点是创建、编辑、检索和管理病人的记录。除了这些核心关注点外, 大型系统还有第二类的功能性关注点。这些包括与核心关注点共享信息的功能, 满足其非功能性需求所需要的功能等。

举个例子, 考虑有这样的一个系统, 它的一个需求是提供对共享缓存的并发访问。一个进程向缓存中输入数据, 另一个进程从同一个缓存读取数据。此共享缓存是数据获取系统的一部分, 生产者进程将数据放入共享缓存, 消费者进程从中将数据取走。在这里核心关注点是维护一个共享缓存, 所以核心功能是向缓存添加和从缓存删除元素。但是, 要确保生产者进程和消费者进程不相互干扰, 必须要有一个第二类的同步关注点。系统设计需要保证生产者进程在数据还没有消费之前不能重写它, 消费者进程不能从空缓存中提取数据。

除了这些第二类关注点外, 其他的关注点如服务质量关注点和机构政策关注点反映的是基本的系统需求。一般来讲, 这些关注点是系统关注点——它们作用于整个系统而不是作用于单个需求或者是对这些需求的程序实现。我们为了和核心关注点相区别, 称这些为横切关注点。第二类功能性关注点也可能是横切关注点, 虽然它们不总是横贯整个系统; 更确切地说, 它们关联于能提供若干相关功能的一组核心关注点。

横切关注点如图 21-1 所示, 此图基于一个网上银行系统的例子。此系统有一些针对新客户的需求, 如信用审查和地址检验。它也有关于已经存在的客户的管理方面的和客户账户管理方面的需求。所有这些都是核心关注点, 因为它们都和系统的主要目标相关(提供网上银行服务)。但是, 此系统也有根据银行信息安全政策的信息安全需求, 以及能够确保在系统失败时数据不至于丢失的恢复需求。这些是横切关注点, 因为它们会影响系统所有其他需求的实现。

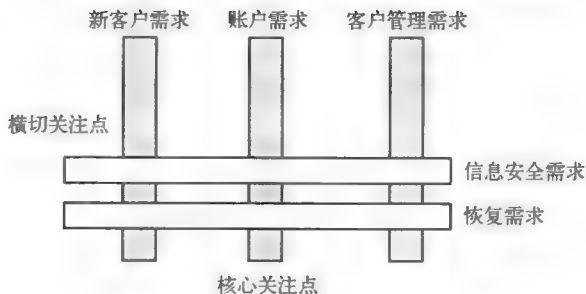


图 21-1 横切关注点

编程语言抽象, 如子程序和类是通常用于组织和构造系统核心关注点的机制。但是, 在传统的编程语言中, 核心关注点的实现总是包含额外的代码来实现横切关注点, 功能性关注点、服务质量关注点和政策关注点。这导致了两种不希望出现的现象——混乱和分散。

当系统中的某个模块包含实现不同系统需求的代码时混乱状态就出现了。图 21-2 的例子是一个有界缓存系统的部分简化了的实现代码, 能说明这个现象。图 21-2 实现了向缓存中添加一项的输入操作。但是, 当缓存是满的, 此算法必须等待直至对应的读取操作从缓冲中移除一项。细节并不重要, 本质问题是使用了 `wait()` 和 `notify()` 调用来同步输入和读取操作的。支持主要关注点(在此情景中, 是把记录放入缓存)的代码, 与实现同步的代码纠缠在一起。与保证互斥的第二类关注点相关联的同步代码需要包含在所有访问共享缓存的方法中。关联于同步关注点的代码如图 21-2 中的阴影代码所示。

```
synchronized void put (SensorRecord rec )
{
    // 检查缓冲中是否有空间；若无则等待
    if ( numberOfEntries == bufsize)
        wait () ;
    // 向缓冲末尾添加记录
    store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
    back = back + 1 ;
    // 如果在缓冲末尾，则下一条记录存储在缓冲的开始
    if (back == bufsize)
        back = 0 ;
    numberOfEntries = numberOfEntries + 1 ;
    // indicate that buffer is available
    notify () ;
} // put
```

图 21-2 缓存管理和同步代码间的混乱

分散现象发生在对一单纯关注点（一个逻辑需求或一组需求）的实现分散在多个程序组件中的时候。在需求实现与第二类功能性关注点或政策性关注点有关联的时候容易发生此类现象。

举例来说，假设某医疗记录管理系统，如 MHC - PMS，由多个组件构成，它们分别是有关管理个人信息、用药、会诊、医学图像，诊断和治疗的组件，这些组件实现了系统的核心关注点：维护病人的记录。可以通过选择那些适合诊所的功能性需求的组件，配置为不同类型的诊所使用的系统。

但是，设想还有一个重要的第二类关注点，它是维护统计信息。医院希望能够记录一些详细的统计信息，包括每月接纳多少病人和出院多少病人，有多少病人死亡，开出多少药物，会诊的原因，等等。这些需求的实现要添加能够使数据匿名化的代码（维护病人的隐私），并将其写入统计数据库。统计组件处理统计数据并产生需要的统计报告。

图 21-3 是对它的说明。这个图给出了可能包含在病人记录系统中的 3 个类的例子以及其中的管理病人信息的一些核心方法。阴影区域中的方法是实现第二类统计关注点所需要的方法。你可以看出这个统计关注点分散在其他的核心关注点中了。

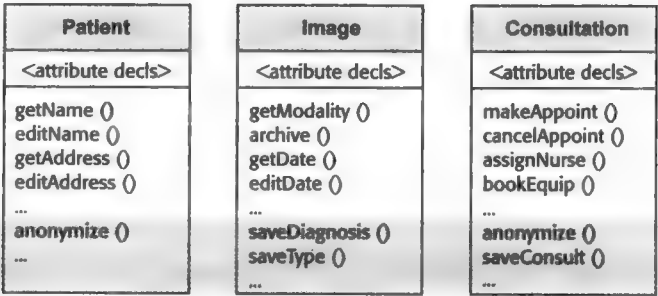


图 21-3 实现第二类关注点方法的分散

当初始系统需求发生改变时分散和混乱的问题就出现了。例如，假定病人记录系统需要收集新的统计数据。对系统的改变不是局限在一个地方，所以我们不得不花时间来寻找系统中的那些需要改变的组件。接着，就要修改每个组件来适应需求的改变。分析组件以及完成测试这些改变需要一定的时间，因此这可能要付出很高的代价。总有可能漏掉了一些需要修改的代码使得统计数据不正确。此外，因为需要做出很多改变，这增加了人为出错和在软件中导入错误的机会。

21.2 方面、连接点和切入点

本节将介绍和面向方面软件开发相关的大部分重要的新概念，并用 MHC-PMS 的例子来说明它们。所用的这些术语是由 AspectJ 的开发者在 20 世纪 90 年代末引入的。但是，这些概念是通用的，并不是 AspectJ 语言所特有的。图 21-4 汇总了读者需要了解的一些关键术语。

术 语	定 义
建议	实现关注点的代码
方面	定义横切关注点的程序抽象。它包括对切入点的定义和与关注点关联的建议的定义
连接点	在执行程序过程中的一个事件，在此处会有与方面相关联的建议的执行
连接点模型	在切入点中可能引用的事件的集合
切入点	在方面中包含的一个语句，定义了连接点，在此相关的方面的建议应该得到执行
编织	通过方面编织器将建议代码合并指定的连接点处

图 21-4 面向方面软件工程中使用的术语

医疗记录管理系统，如 MHC-PMS，包括多个不同的组件，它们分别处理逻辑相关的病人信息。病人组件维护病人的个人信息，药物组件保存可能开出的药物信息，等等。通过基于组件的方法设计此系统，可以配置系统的不同实例。比如，对每个诊所类型可以配置这样的一个版本，医生只允许开与本诊所有关的药物处方。这使得诊所人员的工作简化，减少了医生开错药方的机会。

但是，这样组织就意味着数据库中的信息必须从系统中许多不同的地方更新，比如病人的信息修改可能发生在以下几种情形下：个人详细信息更改的时候，也可能发生在他们所使用的药物改变的时候，或者是发生在给他们指派新医生的时候。为简单起见，假设系统中的所有组件使用一致的命名规则，所有数据更新实现为一些以“update”开头的方法。因此系统中的方法如下所示：

```
updatePersonalInformation (patientId, infoupdate)
updateMedication (patientId, medicationupdate)
```

病人通过病人标识 patientId 来识别，变更封装在第二个参数中，变更的编码细节对此例并不重要。登录系统的医院工作人员负责更新信息。

假设发生了信息安全问题，病人信息遭到恶意修改。可能某人在正常登录后意外忘记退出而使得未经授权人员获得了访问系统的机会。另外，有正当权限的内部工作人员也可能去恶意修改病人的信息。为了减少这种事情再次发生的可能性，引入了一种新的信息安全策略。在对病人数据库修改之前，请求做修改操作的人一定要重新向系统认证他们自己的身份。进行修改的人的详细信息也被单独记录在一个文件中。这有助于再次发生问题时能够轻松追踪。

执行这种新策略的一个办法是修改每个组件中的 update 方法使之调用其他用来认证和登录的方法。或者是，修改系统，使得每次调用更新方法之前，增加调用进行认证的方法，并在改变发生之后记录相关信息。但这两种方法都不是解决问题的非常好的方法。

1. 第一种方法导致了一个混乱的实现。逻辑上讲，更新数据库，认证实现更新的人以及做更新的详细信息的日志是独立的、不相关的关注点。你可能希望在系统的任何地方包含认证而不要日志，或者会希望对除了更新行为之外的其他行为也给出日志。相同的认证和日志代码就需要在不同的方法中包含多遍。

2. 第二种方法导致了一个分散的实现。如果你在更新方法调用之前明确包含认证的调用,并在更新方法调用后显式地包含日志方法调用,那么代码就会包含在系统的许多不同的地方。

认证和日志横贯系统的核心关注点,并可能不得不包含在多个不同的地方。在面向方面的系统中,我们可以把这些横切关注点表达为独立的方面。一个方面包含一个描述,说明将在何处把横切关注点编织进程序,还包含实现关注点的代码。如图 21-5 所示,定义了一个认证方面。本例中使用的符号与 AspectJ 中的风格是一致的,但也简化了它的句法,使之在没有 Java 或 AspectJ 的知识的情况下也是可以理解的。

```

aspect authentication
|
|   before: call (public void update* (...)) // this is a pointcut
|   |
|   |   // this is the advice that should be executed when woven into // the
|   |   // executing system
|   |   int tries = 0 ;
|   |   string userPassword = Password.Get ( tries ) ;
|   |   while (tries < 3 && userPassword != thisUser.password ( ))
|   |   |
|   |   |   // allow 3 tries to get the password right
|   |   |   tries = tries + 1 ;
|   |   |   userPassword = Password.Get ( tries ) ;
|   |   |
|   |   |
|   |   if (userPassword != thisUser.password ( )) then
|   |   |   //if password wrong, assume user has forgotten to logout
|   |   |   System.Logout (thisUser.uid) ;
|   |   |
|   |   |
|   |   } // authentication
|

```

图 21-5 身份认证方面

方面完全不同于其他的程序抽象,因为方面本身包含一个描述,说明它应该在哪里执行。其他的抽象,比如说方法,在抽象的定义和它的使用之间有一个清晰的分离。我们不能通过检查一个方法来得知它会在哪里被调用。可以在方法的有效范围内的任何地方调用一个方法。相对来讲,方面包含一个切入点——一个语句定义方面将在何处被编织进程序。

在本例中,切入点是一个简单的语句:

```
before: call (public void update* (...))
```

这个语句的意思是在执行任何名字以 `update` 开头的方法之前,执行切入点定义之后的代码。字符 `*` 称为“通配符”,可以匹配任何允许的标识符组成的字符串。要执行的代码称为“建议”,它是对横切关注点的实现。这些情形中,“建议”从请求对数据库修改的人那里获得密码并与现在的登录人的密码进行核对。如果不符合,那么用户被强制退出,更新不会进行。

定义和使用切入点,即指定代码应该在哪里执行,是方面的一个显著特征。但是,要理解切入点含义,还需要理解另一个概念——连接点思想。连接点是在程序执行期间发生的一个事件。所以,它可以是一个方法调用,可以是变量初始化,或者是更新一个域,等等。

在程序执行期间会有许多种类型的事件发生。连接点模型定义一个可以在面向方面程序中引用的事件集合。连接点模型没有标准化,每个面向方面编程语言都有自己的连接点模型。例如,在 AspectJ 中,连接点模型中的一部分事件包括:

- 调用事件,调用一个方法或一个构造函数;
- 执行事件,执行一个方法或一个构造函数;
- 初始化事件,类或对象的初始化;
- 数据事件,对域的访问或更新;

异常事件，异常的处理。

切入点识别特别事件（如对指定子程序的调用），“建议”应与这些事件相关联。即可以根据所支持的连接点模型在许多不同的上下文中将“建议”编织到程序中。

1. “建议”可以包含在特定方法执行之前、一组指定的方法序列执行之前，或者是一组其名字与模式描述相匹配（如 `update *`）的方法序列执行之前。

2. “建议”可以包含在某个方法正常或意外返回之后。在图 21-5 的例子中，我们可以定义切入点使之在所有的更新方法调用之后执行日志代码。

3. “建议”可以包含在对象中的一个域被修改时。我们可以包含“建议”来监视或修改那个域。

将“建议”包含在切入点中所定义的连接点上方面编织器的责任。方面编织器是编译器的扩展，它处理构成系统的方面、对象类和方法的定义。然后编织器通过在指定的连接点上包含方面来产生新的程序。方面得以集成，这样在最终的系统中的正确位置处执行横切关注点。

图 21-6 说明了对身份认证和日志的方面编织，这些方面都是应该包括进 MHC-PMS 中的。有 3 种不同的方法来编织方面：

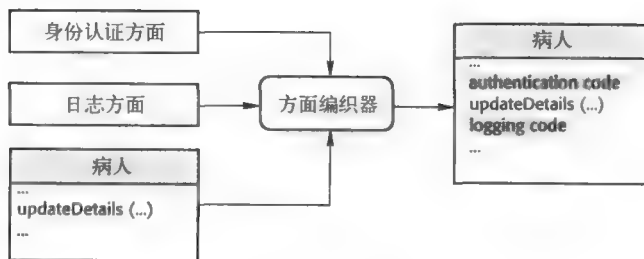


图 21-6 方面编织

1. 源代码预处理，即编织器接收输入的源代码并生成新代码，新代码可以是 Java 或 C++ 这类语言的代码，是能使用标准编译器编译的。这个方法已经在 AspectX 语言中得到采用，其相应的编织器是 XWeaver（Birrner 等，2005）。

2. 链接时编织，即修改编译器使之包括一个方面编织器。处理如 AspectJ 这样的面向方面的语言，并生成标准的 Java 字节码。然后就可以直接由 Java 解释器执行，或经过更进一步的处理生成本地机器码。

3. 执行时动态编织。在此情形中，连接点被监控，每当由切入点所引用的事件发生时，就将相应的“建议”集成到执行程序中。

方面编织中最广泛使用的方法是链接时编织，因为它允许方面高效实现而无大的运行时开销。动态编织是最灵活的方法，但会在程序执行时导致严重的性能损失。源代码预处理目前已经很少使用了。

21.3 采用方面的软件工程

方面最早是作为一个编程语言的结构引入的，但是，正如前面曾讨论过的，关注点的概念实际上是从系统需求中导出的。因此，在系统开发过程的所有阶段采用面向方面的方法是有意义的。在软件工程的早期阶段，采用面向方面的方法意味着利用分离关注点的概念作为考虑需求和系统设计的基础。识别关注点和对关注点的建模应该是需求工程和设计过程的一部分。面向方面的程序语言提供技术支持在你的系统实现中维护关注点的分离。

设计一个系统时，Jacobsen 和 Ng（2004）建议：我们应该让系统支持不同的信息持有者的

关注点，把系统看成是一个核心系统加若干扩展。图 21-7 中使用 UML 包来表示核心系统和扩展。核心系统是实现系统关键目标的一组系统特征。因此，如果特定系统的目标是维护医院中病人的信息，那么此核心系统就提供创建、编辑、管理和访问病人记录数据库的方法。对核心系统的扩展反映额外的信息持有者的关注点，这些关注点必须与核心系统整合在一起。例如，医疗信息系统维护病人信息的私密性是很重要的，所以可能需要一个扩展是关于访问控制的，另外还要有一个是关于加密的，等等。

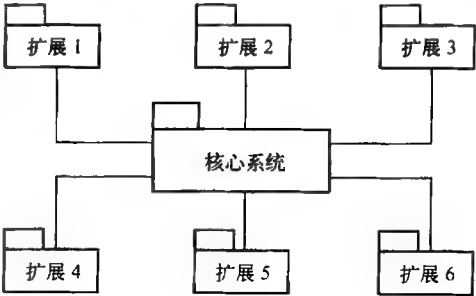


图 21-7 带扩展的核心系统

这里有几种不同类型的扩展，它们源自于 21.1 节中讨论的各种类型的关注点。

- 1. **第二类功能性扩展** 它们在核心系统所提供的功能基础上增加额外的能力。以 MHC-PMS 为例，生成上个月的药物处方的报告就是对病人信息系统的第二类功能性扩展。
- 2. **政策扩展** 它们增加功能性能力来支持某些机构政策。用来增加信息安全特征的扩展就是政策扩展的例子。
- 3. **QoS 扩展** 它们增加功能性能力来帮助获得服务需求的质量，这些服务需求是系统需求中指定了的内容。例如，一个扩展可能是提供缓存以减少数据库访问的次数，或提供自动备份以便当系统崩溃事件发生时能从中恢复。
- 4. **基础结构扩展** 它们增加功能性能力来支持系统在某些特殊执行平台上的实现。比如，在病人信息系统中，基础结构扩展可以用来实现到基础数据库管理系统的接口。修改相关的基础结构扩展就可以更改这些接口。

扩展总是向核心系统添加某类功能或者额外的特征。方面是实现这些扩展的方法，它们还可以在面向方面的编程环境中利用编织工具与核心系统功能相组合。

21.3.1 面向关注点的需求工程

正如 21.1 节中所提出的，关注点反映了信息持有者的需求。这些关注点可能反映了信息持有者所要求的功能需求、系统服务的质量、机构政策或者是与系统总体特性相关的一些问题。因此，为需求工程采用一种方法去识别和指定不同信息持有者的关注点很有意义。“早期方面”这个术语曾用来指在软件生命周期的早期阶段对方面的使用，在此阶段强调的是关注点的分离。

人们认识到在需求工程期间分离关注点的重要性已经有好多年了。表现系统不同角度的视点概念是分离不同信息持有者关注点的一种方法，而且它已经融入到很多需求工程的方法中了（Easterbrook 和 Nuseibeh, 1996；Finkelstein 等, 1992；Kotonya 和 Sommerville, 1996）。这些方法分离不同信息持有者的关注点。视点反映不同的信息持有者群体所要求的独特功能。

但是，也有像图 21-8 所示的那样的横贯所有视点的需求。此图说明，视点可能是不同类型的，但横贯关注点（比如规章、可依赖性、信息安全性）产生可能影响系统中的所有视点的需求。这是作者在开发 Pre-View 方法中所做的工作主要考虑的事情（Sommerville 和 Sawyer, 1997，Sommerville 等, 1998），此方法包括识

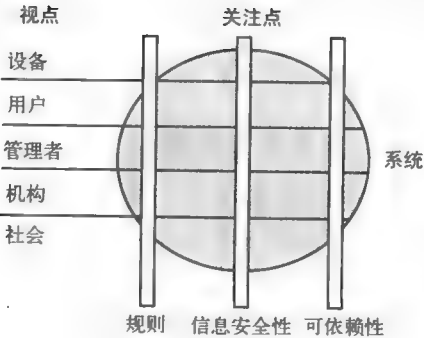


图 21-8 视点和关注点

别横切关注点和非功能性关注点的步骤。

为了开发一个如图 21-7 中风格的系统,就应该识别出核心系统的需求和系统扩展的需求。需求工程的面向视点的方法,即每个视点代表一组信息持有者的需求,是分离核心关注点和第二类关注点的方法。如果我们根据信息持有者的视点来组织需求,接下来就可以通过对它们的分析来发现那些出现在所有或者大部分视点中的需求。这些需求代表了系统的核心功能。其他的视点需求可能是该视点的特有的,可以作为对核心功能的扩展加以实现。

例如,设想你正在开发一个在应急服务中使用的专用设备做跟踪的软件系统。此设备放置在不同的地方,可能存在于一个地区或一个州的很多地方。在紧急事件如洪水或地震出现的时候,应急服务使用此系统来发现离问题发生地最近的可用设备。图 21-9 给出了一个有 3 个可能视点的系统的概要需求。

- | |
|--|
| <p>1. 应急服务用户</p> <p>1.1 寻找一个特殊类型设备 (例如 起重装置)</p> <p>1.2 在指定仓库查看可用的设备</p> <p>1.3 对设备付账</p> <p>1.4 对设备登记</p> <p>1.5 安排将设备运输到应急地点</p> <p>1.6 提交损坏报告</p> <p>1.7 寻找离应急地点最近的仓库</p> <p>2. 应急规划者</p> <p>2.1 寻找一个特殊类型的设备</p> <p>2.2 在指定地点查看可用设备</p> <p>2.3 从仓库登记/检查设备</p> <p>2.4 将设备从一仓库移到另一个仓库</p> <p>2.5 订购新设备</p> <p>3. 维护人员</p> <p>3.1 为维护登记/检查设备</p> <p>3.2 在每个仓库查看可用设备</p> <p>3.3 寻找某个特殊类型设备</p> <p>3.4 对某个设备项查看维护进度</p> <p>3.5 为设备项完善维护记录</p> <p>3.6 显示某仓库中需要维护的所有设备项</p> |
|--|

图 21-9 设备库存系统中的视点

从这个例子中你可以看出所有不同视点的信息持有者都需要能够找到特殊的设备项,浏览每个地点的可用设备并从仓库登记借出设备。因而这些是核心系统的需求。第二类需求支持每个视点的更多的特殊需要。对于系统扩展来说,有支持设备使用、管理和维护这样一些第二类需求。

通过视点所识别出来的第二类功能性需求不一定横贯来自其他视点的需求。例如,只有维护视点关注记录维护操作的完成。这些需求反映了该视点的需要,那些关注点可能无需与其他视点共享。然而,除此第二类功能性需求之外,还有横切关注点,产生对部分或所有视点都很重要的需求。这些通常反映的是作用于整个系统的服务需求的政策和质量。如第 4 章中讨论过的,这些属于非功能性需求,比如信息安全、性能和成本的需求。

在设备库存系统中,横切关注点的一个例子是系统可用性。紧急情况的发生可能很少或者根本没有警告。为了挽救生命可能要求尽可能快地部署好重要设备。因此设备库存系统的可靠性需求就会包括高要求的系统可用性需求。图 21-10 给出了一些可依赖性需求的例子以及相应的解释。通过这些需求,我们就可以判断出对一些核心功能的扩展,这些核心功能包括用于事务日志和状态报告,这使得找出问题和切换到备用系统变得很容易。

需求工程过程的成果应该是一组需求。这个需求集合被划分到核心系统和扩展组件两概念对应的子集中。举例来说,在库存系统中,核心需求的例子如:

- C.1 系统应该允许授权用户查看应急服务库存中的任何设备项的描述。
- C.2 系统应该包括一个搜索工具，允许授权用户搜索单个库存或全部库存中的某个特殊设备项或特殊设备类型。

系统也应该包含一个扩展用于支持设备的获得和替换。因此，对此扩展的需求可能是：

E1.1 为了更换设备，经授权的用户应该能够向可靠厂商下设备订单。

E1.1.1 某个设备项已经订购，它应位于特定的仓库并在此库存中被标记为“已预订”。

- | | |
|--------|-----------------------------------|
| AV.1 | 应该有一个紧急备用系统，与主系统在地理上适当分离。 |
| 解释： | 紧急事件可能影响系统的基本位置。 |
| AV.1.1 | 所有的事务应该在主系统站点和远处的备用站点都有记录。 |
| 解释： | 因而允许事务重放并保持系统数据库的一致。 |
| AV.1.2 | 系统每5分钟应向紧急事件控制室系统发送状态信息 |
| 解释： | 如果主系统不可用，紧急事件控制室系统的操作者能够将其切换到备用系统 |

图 21-10 设备库存系统的可用性相关需求



视点：

作者在第4章中介绍了视点的概念，在那儿解释了视点是如何在构建不同信息持有者需求中使用的。利用视点，你可以确定每组信息持有者对核心系统的需求。

<http://www.SoftwareEngineering-9.com/Web/Requirements/Viewpoints.html>

作为一般规则，你要避免让系统有太多的关注点和扩展。这些太多的关注点和扩展只能让读者混淆并可能导致不成熟的设计。这一点限制了设计者的自由，并有可能导致系统的设计不能满足服务需求的质量。

21.3.2 面向方面的设计和编程

面向方面的设计是利用方面进行系统设计的过程，通过方面来实现那些在需求工程过程中所找出来的横切关注点和扩展。在此阶段，我们需要把与所要解决的问题相关联的关注点翻译成程序中对应的方面。我们也需要了解如何将 these 方面与系统的其他组件组合在一起，并确保不发生组合的二义性。

需求的高层声明为识别某些系统扩展提供了基础，这些扩展可以实现为方面。接下来，就需要给出更详细的需求，从而识别出更多的扩展并了解所需要的功能。其中一种方法是找出一组和每个视点相关的用例（在第4、5章中论述过）。用例模型关注交互，比用户需求更具体。你可以把它看做是需求和设计之间的桥梁。在用例模型中，我们描述用户交互的步骤，从而开始识别和定义系统中的类。

Jacobsen 和 Ng (2004) 出过一本书，讨论用例如何在面向方面的软件工程中使使用。他们建议每个用例代表一个方面，他们也提议扩展用例方法来支持连接点和切入点。另外也引入了用例片段和用例模块的概念，用例片段和用例模块包含了用于实现方面的类片段，通过它们的组合可以创建出完整的系统。

图 21-11 示意的是包括3个用例的库存管理系统的一部分。这些用例反映了往库存中增加设备和订购设备的关注点。设备订购和往仓库中增加设备是相关的关注点。一旦所订购的项交付，它们必须被加入到库存中并移交到某个设备仓库。

UML 已经包含了扩展用例的概念。扩展用例是对另一个用例的功能扩充。图 21-12 示意的是

如何用下设备订单这个用例来扩展增加设备到特定仓库这个核心用例的。如果要添加的设备不存在，那可以订购，当货物到达时添加到仓库中。在用例模型开发过程中，我们应该寻找共性特征。如果可能的话，将用例设计成核心用例加扩展用例的形式。横切特征，如对所有交易的日志，也可以表示为扩展用例。Jacobsen 和 Ng 讨论了如何将此类型的扩展实现为方面的问题。

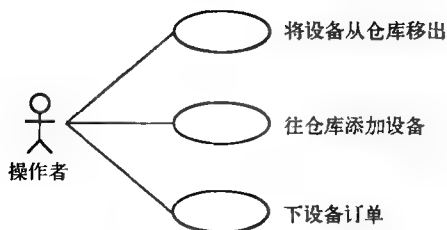


图 21-11 来自库存管理系统的用例

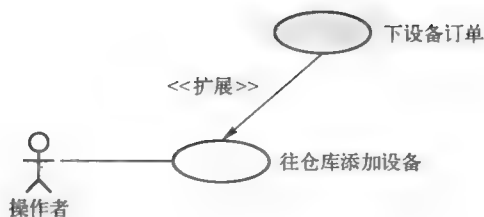


图 21-12 扩展用例

如果我们接受和使用面向方面的设计，那么给出一种高效的面向方面的设计过程就是非常重要的了。建议面向方面的设计过程应包括如图 21-13 所示的一些活动，这些活动是：

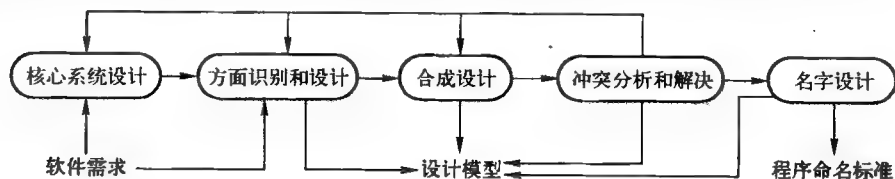


图 21-13 通用的面向方面设计过程

1. **核心系统设计** 在此阶段，设计一个能支持系统核心功能的体系结构。该体系结构也必须考虑服务需求的质量，比如性能和可依赖性需求。

2. **方面识别和设计** 从系统需求中得到的扩展开始，通过对它们的分析判断它们本身是否构成方面，或者是应该将它们剖分成几个方面。一旦找出了方面，就可以分别对它们进行设计了，同时兼顾对核心系统特征的设计。

3. **合成设计** 在此阶段，分析核心系统设计和方面设计，从中发现在哪里应该将方面与核心系统合成在一起。重要的是，找到程序中将方面编织进去的连接点。

4. **冲突分析和解决** 方面的问题在于当它们与核心系统合成在一起时容易发生互相干扰。当有一个切入点 and 若干个指定要在此处合成的方面的时候冲突就发生了。然而，还可能有更微妙的冲突发生。因为方面是独立设计出来的，它们可能都对核心系统功能有个假设，而这些核心功能都是要修改的。当许多方面合成在一起的时候，一个方面对系统功能的影响可能是其他方面所无法预料到的。因而整个系统的行为会与期望的有所不同。

5. **名字设计** 这是一项很重要的活动，它定义程序中实体命名的标准。这对于避免意外切入点问题是必要的。意外切入点发生在这样的情况下，在某些程序连接点上，名字意外匹配了切入点模式中的名字，因此建议会在此时无意中被应用。显然这不是我们所希望的，程序也会有非预期的行为。因此，你要设计一个命名方案将这种情况发生的可能性降到最低。

这个过程自然地是个迭代过程，首先是有一个初始的设计建议，然后在分析和理解设计问题的基础上不断地细化它。通常情况，我们会将需求中找出的扩展细化为很多个方面。

面向方面设计过程的输出结果是一个面向方面的设计模型。这可以用 UML 扩展版本来表述，此版本包括新的方面特有的结构，如由 Clarke 和 Baniassad (2005) 以及 Jacobsen 和 Ng (2004)

提议的等。“方面 UML”的基本元素包括一些方面建模的手段和指定连接点的手段，此连接点既是方面建议应和核心系统组合在一起的地方。

图 21-14 是一个面向方面的设计模型的例子。在此使用的是 Jacobsen 和 Ng 所提出的方面的 UML 约定。图 21-14 示意了应急服务库存的核心系统和需要与核心组合的一些方面。前面已经给出了一些核心系统类和某些方面。这是一个简化的描述，一个完备的模型应该包含更多的类和方面。注意是如何使用 UML 注释来提供有关被某些方面所横贯的一些类的额外信息。

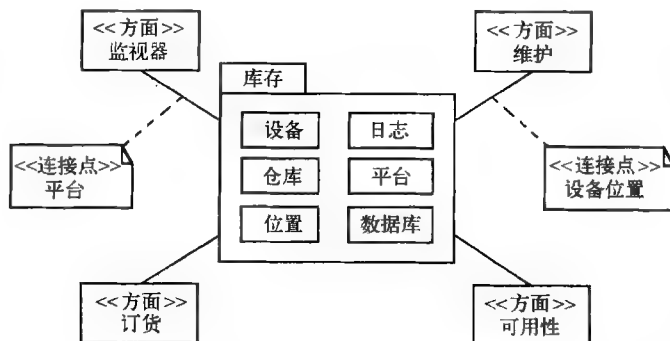


图 21-14 面向方面的设计模型

图 21-15 是方面更具体的模型。很显然，在设计方面之前，需要有个核心系统的设计。因为这里无法展开陈述，所以对核心系统中的类和方法进行了诸多假设。

方面的第一部分宣布切入点，指定它将在哪与核心系统组合。例如，第一个切入点指定方面会在调用 `getItemInfo(..)` 这个连接点处组合进系统。接下来的部分定义将由方面所实现的扩展。在此例中，扩展声明可以写成：

“在方法 `viewItem` 中，对方法 `getItemInfo` 的调用后，应该包含进去一个对方法 `displayHistory` 的调用，以显示维护记录。”

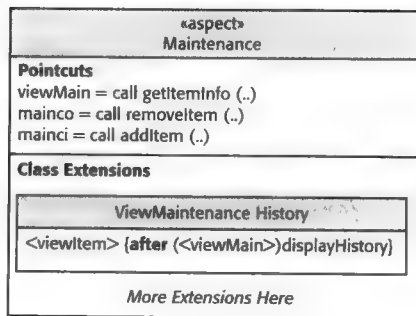


图 21-15 方面模型的部分内容

面向方面的编程（AOP）是伴随着编程语言 AspectJ 的发展由 Xerox 公司的 PARC 实验室于 1997 年首先开始的。AspectJ 依然是应用最广泛的面向方面的语言，但是其他语言（比如 C# 和 C++）也实现了面向方面的扩展。其他实验性语言也已经开发出来支持显式地分离关注点和组合关注点。同时，.NET 框架中也有对 AOP 的尝试性实现。面向方面的编程在其他的书中有全面的介绍（Colyer 等，2005；Gradecki 和 Lezeiki，2003；Laddad，2003b）。

如果要采用面向方面的方法来设计你的系统，就必须先找出核心功能和扩展，将扩展实现为横切的方面。于是编程过程的焦点将是编写代码实现核心系统和扩展功能。紧要的是，在方面中定义切入点，这样方面建议就能够在正确的地方被编织进基础代码中。

正确地指定切入点是非常重要的，因为它们定义了方面的建议在哪里和核心功能结合起来。如果在切入点的描述上犯错误，那么方面的建议将在错误的地方被编织到程序中。这将导致不希望的也是不可预料的程序行为。坚持在系统设计过程中遵循所建立的命名标准是必要的。同时也要复查所有的方面以保证如果有两个或多个方面在相同的连接点上编织进核心系统时没有冲突发生。一般来说，最好能够完全避免，但是有时候，将它们实现为一个关注点可能是最好的

方法。在这种情况下，就必须保证方面是完全独立的。程序的行为不应依赖于方面编织进程序的顺序。

21.3.3 检验和有效性验证

正如第8章中所论述的，检验和有效性验证是证明一个程序符合它的描述（检验）以及符合它的信息持有者的真正需要（有效性验证）的过程。静态检验技术集中于手动或自动分析程序的源代码。动态验证或测试是试图发现程序中的缺陷或证明程序满足它的需求。如果缺陷检测作为目标的话，测试过程会在程序源代码的知识指导下进行。测试覆盖的度量给出源代码语句执行的测试的有效性。

对于面向方面的系统，其有效性验证测试的过程和对其他系统没有什么区别。最终执行的程序被看做是一个黑盒，设计测试以说明系统是否符合它的需求。然而，当通过程序的源代码找可能的缺陷测试时，方面的使用会导致程序检查和白盒测试出现实际问题。

如第24章中所论述的，程序检查就是一组读者阅读程序的源代码寻找编程人员所引入的缺陷。它是一个非常有效的发现缺陷的技术。但是，面向方面的程序是不能顺序地阅读的（例如自上而下的阅读）。因此它们更难以理解。

程序的可理解性的一般准则是读者能够自左向右、自顶向下地阅读程序而不用切换注意力到其他代码部分。这使得读者容易阅读也使得编程人员犯错误的机会小得多，因为注意力集中在一段代码上。改善程序的可读性是引入结构化编程（Dijkstra 等，1972）和从高级语言中除去无条件转移（go-to）语句的主要原因。

在面向方面的系统中，顺序地阅读代码是不可能的。阅读者必须检查每个方面，了解它的切入点（它可能是模式）和面向方面语言的连接点模型。当阅读程序时，必须识别每个潜在的连接点并把注意力转移到方面代码，看它是否在那点上编织的。然后，注意力接着回到基代码的主控制流。现实中，这在认知上是不可能的，检查面向方面程序的唯一可能的方法是使用代码阅读工具。

代码阅读工具可以使面向方面的程序“变平”，展现在读者面前的是已经将方面在指定的连接点处编织进程序的代码。但是，这并不是针对代码阅读问题的完整解决方案。一个语言中的连接点模型可能是动态的而非静态，并且不太可能证明扁平化的程序能不折不扣地按照要执行的程序那样做。而且，因为不同的方面有相同切入点描述是可能的，程序阅读工具必须了解方面编织器是如何处理这些“竞争”方面的以及如何安排其组成的。

白盒或结构化测试是系统化测试方法，使用程序源代码知识去设计缺陷测试。其目标是设计能提供一定程度的程序覆盖的测试。典型的，测试集应该保证程序中的每个逻辑路径都执行过，使得每个程序语句都至少执行过一次。程序执行分析器用来证明能达到这个程度的测试覆盖。

在面向方面的系统中，此方法有两个问题：

1. 程序代码的知识如何能用来系统地导出程序测试？
2. 测试覆盖的真正含义是什么？

对没有无条件分支语句的结构化程序设计测试（例如对某个方法代码的测试），可以导出程序流程图，流程图显示了贯穿程序的每个逻辑执行路径。接着，可以检查代码，针对流程图中的每个路径，选择使路径能够执行的输入。

但是，面向方面的程序不是结构化的程序。控制流被“come from”语句所中断（Constantinos 等，2004）。在某些基代码执行中的连接点上，某个方面可能得到执行。因为不确定在这种情况下是否可以构造一个程序流程图，所以，用系统化的设计程序测试来保证所有的基代码和方面

的组合都得到执行是很困难的。

在面向方面的程序中，还有一个问题，那就是确定测试覆盖的内涵。它意味着每个方面的代码至少执行一次？这是一个非常弱的条件，因为方面和基代码之间在方面被编织的连接点处有交互。那么，覆盖测试的概念是否需要扩展为使得方面的代码在每个切入点指定的所有连接点上至少执行一次？在这种情况下，如果不同方面定义了相同的切入点又会发生什么现象？这些既包含理论上的问题也包含实践上的问题。我们需要工具来支持面向方面的程序测试，它能够帮助评估对系统的测试覆盖的程度。

如第 24 章中所论述的，大型项目通常都有单独的质量保证队伍，他们制定测试标准并要求严格，保证程序复查和测试完全按照这些标准来完成。对于面向方面程序来说，在检查和导出测试方面所遇到的问题成为在这样的大型软件项目中采用面向方面的软件开发方法的巨大障碍。

与程序检查和白盒测试中的问题一样，Katz (2005) 指出在面向方面编程中测试环节存在的其他一些问题：

1. 应该如何定义方面以便能从中推出对方面的测试？
2. 相对于所要编织到的基系统，如何对方面进行独立测试？
3. 如何测试方面的干涉？如前面曾论述的，当两个或多个方面使用相同的切入点描述时方面干涉就会发生。
4. 如何设计测试，使得所有程序中的连接点都执行到，且适当的方面测试得到实施？

从根本上讲，这些测试问题的发生是因为，方面是与系统的主体代码紧凑集成的而不是松散集成的，因此它们很难孤立进行测试。因为它们可能在很多不同的地方编织进程序，我们就不能肯定在一个连接点能成功工作的方面在其他连接点处也能正常工作。所有这些都是面向方面的软件开发所需要进一步研究的问题。

要点

- 面向方面方法对软件开发的最主要的好处是它支持对关注点的分离。通过用方面来表示横切关注点，单个关注点才能在不修改程序其他部分的前提下被理解、复用和修改。
- 混乱现象发生在系统中一个模块包含实现不同系统需求的代码的情形。相对分散现象发生在系统中的单个关注点被分散到程序中的几个组件中实现的时候。
- 方面，包括一个切入点和一个建议。切入点，即一个定义在什么地方将方面编织进程序的声明，建议，即实现横切关注点的代码。连接点是在切入点中查到的事件。
- 为了确保对关注点的分离，系统可以设计成核心系统和扩展两部分，核心系统实现信息持有者的主要关注点，而扩展用来实现第二类关注点。
- 要识别关注点，可以将面向视点的方法用于需求工程，从而导出信息持有者的需求并识别出横贯的服务质量关注点和政策关注点。
- 从需求到设计的转换可以通过找出用例来实现，每个用例代表一个信息所有者关注点。设计可以使用带有方面模板的 UML 扩展版来建模。
- 对面向方面程序来讲，在检查和导出测试上的问题是在大型软件项目中采用面向方面的软件开发方法的巨大障碍。

进一步阅读材料

《Aspect-oriented programming》此 CACM 的特刊中有多篇文章是适合一般读者的，是阅读面向方面编程的一个好的入门读物 (Comm. ACM, 44 (10), October 2001)。http://dx.doi.org/10.1145/383845.383846。

《Aspect-oriented Software Development》这是一个论文集，收录了面向方面的软件开发各个方面的文章，其作者大都是本研究领域中的顶级研究人员（R. E. Filman, T. Elrad, S. Clarke and M. Aksit, Addison-Wesley, 2005）。

《Aspect-oriented Software Development with Use-Cases》这是一本面向软件设计人员的实用教材，讨论了如何使用用例来管理关注点的分离并用它作为面向方面设计的基础（I. Jacobsen and P. Ng, Addison-Wesley, 2005）。

练习

- 21.1 在大型系统中会有哪些不同类型的项目信息所有者关注点？方面如何能支持对每个类型的关注点的实现？
- 21.2 总结一下什么是混乱和分散。举例说明为什么当系统需求改变时，会产生混乱和分散问题。
- 21.3 连接点和切入点有什么不同？解释它们是怎样作用于将代码编织到程序中以处理横切关注点的。
- 21.4 用实现本质需求的核心系统加上实现附加功能的扩展来组织系统的思想是基于什么假设的？你能思考一下不能恰当运用此模型的系统吗？
- 21.5 为 MHC - PMS 的开发需求描述时，应该考虑什么视点？哪些可能成为最为重要的横切关注点？
- 21.6 使用图 21-9 中的每个视点的功能概要，为设备库存系统找出除了图 21-11 中以外的其他 6 个用例。在适当的地方，说明如何将其中一些做成扩展的用例。
- 21.7 利用图 21-15 中的方面原型的符号，给出图 21-14 中所示的订购和监控两个方面的详细内容。
- 21.8 解释方面干涉为什么会发生，并给出在系统设计过程中应该做哪些工作来减少方面干涉的问题。
- 21.9 解释为什么用模式作为表达切入点描述增加了测试面向方面程序的问题。为了回答这个问题，请思考程序测试大都是如何利用期待输出与实际系统输出的比较的。
- 21.10 对如何使用方面来简化对程序的调试。

参考书目

Birrer, I., Pasetti, A. and Rohlik, O. (2005). 'The XWeaver Project: Aspect-oriented Programming for On-Board Applications'. <http://control.ee.ethz.ch/index.cgi?page=publications;action=details;id=2361>.

Clark, S. and Baniassad, E. (2005). *Aspect-Oriented Analysis and Design: The Theme Approach*. Harlow, UK: Addison-Wesley.

Colyer, A. and Clement, A. (2005). 'Aspect-oriented programming with AspectJ'. *IBM Systems J.*, 44 (2), 301-8.

Colyer, A., Clement, A., Harley, G. and Webster, M. (2005). *eclipse AspectJ*. Upper Saddle River, NJ: Addison-Wesley.

Constantinos, C., Skotiniotis, T. and Stoerzer, T. (2004). 'AOP considered harmful'. *European Interactive Workshop on Aspects in Software (EIWAS'04)*, Berlin, Germany.

Dijkstra, E. W., Dahl, O. J. and Hoare, C. A. R. (1972). *Structured Programming*. London: Academic Press.

Easterbrook, S. and Nuseibeh, B. (1996). 'Using ViewPoints for inconsistency management'. *BCS/IEE Software Eng. J.*, 11 (1), 31-43.

Finkelstein, A., Kramer, J., Nuseibeh, B. and Goedicke, M. (1992). 'Viewpoints: A Framework for Integrating Multiple Perspectives in System Development'. *Int. J. of Software Engineering and Knowledge Engineering*, 2 (1), 31-58.

Gradecki, J. D. and Lezeiki, N. (2003). *Mastering AspectJ: Aspect-Oriented Programming in Java*. New York: John Wiley & Sons.

Jacobson, I. and Ng, P-W. (2004). *Aspect-oriented Software Development with Use Cases*. Boston: Addison-Wesley.

Katz, S. (2005). 'A Survey of Verification and Static Analysis for Aspects'. <http://www.aosd-europe.net/documents/verificM81.pdf>.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G. (2001). 'Getting Started with AspectJ'. *Comm. ACM*, **44** (10), 59–65.

Kotonya, G. and Sommerville, I. (1996). 'Requirements engineering with viewpoints'. *BCS/IEE Software Eng. J.*, **11** (1), 5–18.

Laddad, R. (2003a). *AspectJ in Action*. Greenwich, Conn.: Manning Publications Co.

Laddad, R. (2003b). *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich, Conn.: Manning Publications.

Sommerville, I. and Sawyer, P. (1997). 'Viewpoints: principles, problems and a practical approach to requirements engineering'. *Annals of Software Engineering*, **3** 101–30.

Sommerville, I., Sawyer, P. and Viller, S. (1998). 'Viewpoints for requirements elicitation: a practical approach'. *3rd Int. Conf. on Requirements Engineering*. Colorado: IEEE Computer Society Press, 74–81.

软件管理

在前面已经提到过，软件工程与其他类型的程序设计的重要不同在于软件工程是一个管理过程，希望读者了解软件开发发生在机构内部并受到进度安排、预算以及其他一些机构约束的控制。因而管理对于软件工程来说是非常重要的。此部分介绍一系列的管理话题，着重介绍技术管理问题，而不是“更软”的人的管理问题或是更战略性的企业管理问题。

第 22 章介绍软件项目管理和它的最重要的风险管理部分。与项目规划一样，风险管理，即管理者识别什么会出现问题以及给出在出现问题时如何做的计划，这是项目管理职责的重要方面。这一章也介绍部分人的管理和团队协作方面的一些内容。

第 23 章介绍项目规划和估计。本章把条形图作为基本规划工具并解释为什么计划驱动的开发将继续是一个重要的开发方法，尽管敏捷方法也很成功。此外，还将讨论影响系统定价的一些问题以及软件成本估计的技术，并使用 COMOCO 成本模型族来描述算法成本建模并解释此方法的优点和缺点。

第 24 ~ 26 章是关于质量管理的问题。质量管理是关于确保和改善软件质量的过程和技术，第 24 章介绍此话题，讨论质量管理标准的重要性，如何在质量保证过程中使用复查和审查技术，以及软件度量在质量管理中的作用。

第 25 章讨论配置管理。这是一个对所有由团队开发的大型系统都很重要的问题。然而，对配置管理的需要对于那些只知道一点个人软件开发的学生来说并不总是很明显的。所以在此描述配置管理的各个方面，包括配置规划、版本管理、系统构建以及变更管理。

最后，第 26 章介绍软件过程改善，即如何修改过程以便产品和过程属性都能够得到改善？本章讨论一般过程改善过程的各个阶段，即过程度量、过程分析和过程变更。然后继续介绍在过程改善中应用 SEI 的基于能力的方法，并简要解释能力成熟度模型。

项目管理

目标

本章的目的是介绍软件项目管理和两项重要的管理活动，即风险管理和人员管理，通过阅读本章，你将了解以下内容：

- 软件项目管理者的主要任务；
- 风险管理的概念以及在软件项目中可能出现的一些风险；
- 理解影响工作动力的因素以及这些因素对软件项目管理者的意义；
- 理解影响团队协作的主要问题，比如团队的构成、团队的组织 and 团队的沟通。

软件项目管理是软件工程的一个重要组成部分。我们需要软件项目管理是因为专业的软件工程总是要受预算和工程进度的制约。软件项目管理者的任务是确保软件项目满足和服从这些约束，并确保交付高质量的软件产品。好的管理不能确保项目成功，但是，不好的管理是注定要带来项目的失败：软件常常不能按期完成，成本是预期的几倍，或者不能满足客户的要求。

项目管理成功的标准对于不同的项目很显然是不同的，但是对于大多数项目来说，最重要的目标是：

1. 在约定的时间将软件产品交付给客户；
2. 将全部成本控制在预算之内；
3. 交付的软件产品满足客户的要求；
4. 保持一个愉悦并且运作良好的开发团队。

这些目标不只是软件工程所独有的，也是所有工程项目的目标。然而，软件工程管理与其他工程管理相比，在很多方面有显著的区别，这样使得软件工程管理具有相当大的难度。以下列出了软件工程管理的一些不同之处：

1. **软件产品是无形的** 对于像造船或其他一般的民用工程项目，其管理者能够看见正在制造的产品。如果进度跟不上，就能从产品的现状看出来，整个结构中的某些部分很明显是未完工的。而软件产品是无形的，看不见摸不着。软件项目管理者不能依靠简单的查看正在开发的产品来了解进展情况。在一定程度上，他们只能依靠其他人提供信息来掌握工作进度。

2. **大型软件项目常常是“一次性的”项目** 大型软件项目在很多方面都不同于早先的项目，因而管理者纵然有通过计划降低不确定性的经验，也很难预见将出现的问题。此外，计算机和通信技术飞速发展，早先的经验很快变得过时了，其中的教训不能在新的项目中发挥作用。

3. **软件开发过程是可变的和机构特定的** 对于某些系统，比如桥梁和建筑等类型的工程过程我们已经了如指掌。然而，软件开发的过程对于不同的机构来说是相当不同的。虽然我们在过程标准化和过程优化方面已经取得了很大的进步，但是我们还不能确切地预见某一软件过程何时有可能出现问题。特别当这个软件项目是某个更大的系统工程项目的一部分时尤为明显。

基于以上这些原因，一些软件项目延期、超出预算、进度慢也就不足为奇了。软件系统往往是全新的而且技术上也有所创新。具有创新性的工程项目（例如，新的运输系统）也经常会存

在进度问题。一旦出现困难,软件项目的按期、按预算完成就会变得几乎不太可能。

软件管理者的工作内容没有一定的标准。开发机构和开发的软件产品决定着管理工作的内容。尽管如此,绝大多数管理者要在某些阶段对下列部分或全部活动负起责任:

1. **项目规划** 项目管理者负责规划、评估以及调度项目开发,并给人员分配任务。他们监督工作确保工作是按要求的标准进行,并且监督开发过程,检查开发是否按时进行以及是否在预算之内。

2. **工作报告** 项目管理者通常有责任向客户和开发此软件的公司的经理汇报项目进展情况。他们必须能够进行各个层次的交流,从详尽的技术细节到概要的管理总结。项目管理者必须能从详细的项目报告中抽出重要的信息,写成简明、一致的文档,并在项目评审时陈述出来。

3. **风险管理** 项目管理者必须评估并且监测某些可能影响项目的风险,并在问题出现时立刻采取行动。

4. **人员管理** 项目管理者负责管理一个团队。他们必须为他们的团队挑选人员并且建立能够使团队高效运作的工作方式。

5. **提出书面建议** 软件项目的第一个阶段可能要写出完成该项目的建议书,在建议书中要写清楚项目的目标和实现该方法的方法,通常还要估算出项目的成本和进度,有时还要说明应该与某一特定的机构或团队签约的理由。建议书的写作非常重要,许多软件公司之所以生存下来是因为他们保持有大量的建议书和合同。写建议书没有固定的格式以供参考,它是一种来自长期实践和经验的技巧。

本章主要介绍风险管理和人员管理。项目规划是一项重要的课题,将在第23章讨论。

22.1 风险管理

能预见可能影响正开发的软件的项目进度或产品质量的风险,并采取行动避免这些风险,是项目管理者的一项重要任务(Hall, 1998; Ould, 1999)。可以把风险看做是一些可能实际发生的不利因素。风险可能危及整个项目、正在开发的软件或者开发机构。这些风险种类可以定义如下:

1. **项目风险** 是影响项目进度或项目资源的风险。比如失去一位经验丰富的设计师。寻找一位经验和技能满足要求的替代者可能需要很长一段时间,而这带来的直接后果就是软件的设计需要更长的时间才能完成。

2. **产品风险** 是影响开发中软件的质量或性能的风险。例如,购买的组件不能达到预期目标。这可能会影响到整个系统的性能,导致整个系统运行速度下降。

3. **业务风险** 是影响软件开发机构或软件产品购买机构的风险。例如,竞争对手推出了新的产品。竞争产品的推出可能预示着先前做出的关于现有软件产品的销售情况可能过于乐观。

当然,这种分类方式会有重叠。如果富有经验的程序设计师离职,由此带来的风险可能是项目风险。即使能立刻找到替代人员,进度也将受到影响。新来的项目成员不可避免要花一些时间来熟悉这个项目,并不能立刻高效地投入工作。因此,系统的交付可能延期。同时这也会是产品风险,因为继任者可能不如前者有经验,因而可能犯错误。最后,它也会是业务风险,因为有经验的程序员是取得新的合同的关键。

项目经理应该将风险分析的结果以及相应的后果,包括对项目的风险后果,对产品的风险后果以及对业务的风险后果,写到项目计划中。有效的风险管理能够使项目管理者对未来出现的问题处理自如,并保证这些风险不会导致不可接受的预算和进度偏差。

项目的风险类型取决于项目本身的特点和软件开发的机构环境。然而有许多共同的风险

不依赖于开发的软件的种类，即风险可能发生在任何项目中。图 22-1 给出了其中的一部分共同的风险。

风 险	风 险 类 型	描 述
职员跳槽	项目	有经验的职员将会未完成项目就跳槽
管理层变更	项目	管理层结构将会发生变化，不同的管理层考虑、关注的事情会不同
硬件缺乏	项目	项目所需的基础硬件没有按期交付
需求变更	项目和产品	软件需求与预期的相比，将会有许多变化
描述延迟	项目和产品	有关主要的接口的描述未按期完成
低估了系统规模	项目和产品	过低估计了系统的规模
CASE 工具性能较差	产品	支持项目的 CASE 工具达不到要求
技术变更	业务	系统的基础技术被新技术取代
产品竞争	业务	系统还未完成，其他有竞争力的产品就已经上市了

图 22-1 常见的项目、产品和业务风险的实例

风险管理对软件项目而言尤为重要，因为绝大多数项目都存在固有的不确定性。这些不确定性产生于：宽泛定义的需求（需求随客户要求的变化而改变），对软件开发所需时间和资源估算的困难，以及项目组成员技术上的差异性。项目管理者应该预见风险，了解这些风险对项目、产品和业务的冲击，并采取措施规避这些风险。可以制定应急计划，这样一旦风险来临，才有可能采取快速防御行动。

图 22-2 是说明风险管理过程的略图，该过程包括以下几个阶段：

- 1. **风险识别** 识别可能的项目风险、产品风险和业务风险。
- 2. **风险分析** 评估这些风险出现的可能性及其后果。
- 3. **风险规划** 制定计划说明如何规避风险或最小化风险对项目的影响。
- 4. **风险监控** 定期对风险和缓解风险的计划进行评估，并随着有关风险信息增多及时修正缓解风险的计划。

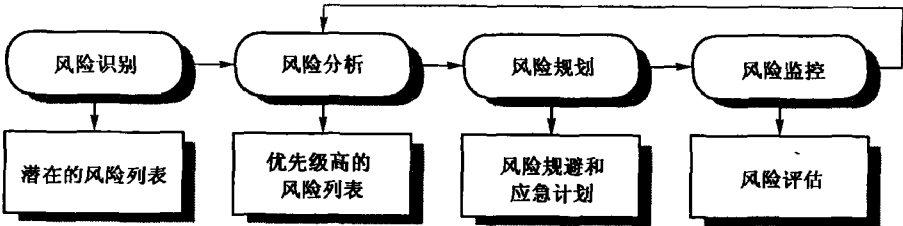


图 22-2 风险管理过程

风险管理过程的结果应该记录在风险管理计划中，具体包括：对项目所面临的风险的讨论，对这些风险的分析，以及当风险可能成为一个现实问题时管理这些风险的建议。

风险管理过程也是一个贯穿项目全过程的迭代进行的过程，从最初的计划制定开始，项目就处于被监控状态以检测可能出现的风险。随着有关风险的信息增多，需要重新进行分析，重新确定风险的优先级。对风险规避和应急计划要进行修正。

22. 1. 1 风险识别

风险识别是风险管理的第一阶段。这一阶段主要是发现可能对软件工程过程、正在开发的

软件或者开发机构产生重大威胁的风险。风险识别可以通过项目组对可能的风险的集体讨论完成。或者只凭借管理者的经验识别最可能或者最关键的风险。

作为风险识别的开始，需要列出可能的风险类型。这些类型包括：

- 1. 技术风险 源于开发系统的软件技术或硬件技术的风险。
- 2. 人员风险 与软件开发团队的成员有关的风险。
- 3. 机构风险 源于软件开发的机构环境的风险。
- 4. 工具风险 源于软件工具和其他用于系统开发的支持软件的风险。
- 5. 需求风险 源于客户需求的变更和需求变更的处理过程的风险。
- 6. 估算风险 源于对构建系统所需资源进行估算的风险。

图 22-3 对上述每一种风险都给出了可能的风险实例。风险识别过程的结果应该是列出一长串可能发生的风险，这些风险可能影响到软件产品、过程或业务。紧接着，需要削减风险列表到便于管理的程度为止。实际工作中是不可能关注过多的风险的。

风 险 类 型	可能的风险
技术	系统使用的数据库的处理速度不够快 要复用的软件组件有缺陷，限制了项目的功能
人员	招聘不到符合项目技术要求的职员 在项目的非常时期，关键性职员生病，不能发挥作用 职员所需的培训跟不上
机构	重新进行机构调整，由不同的管理层负责这个项目 开发机构的财务出现问题，必须削减项目预算
工具	CASE 工具产生的编码效率低 CASE 工具不能被集成
需求	需求发生变化，主体设计要返工 客户不了解需求变更对项目造成的影响
估算	低估了软件开发所需要的时间 低估了缺陷的修补率 低估了软件的规模

图 22-3 不同风险类型实例

22.1.2 风险分析

在进行风险分析时，要逐一考虑每个已经识别出的风险，并对风险出现的可能性和严重性做出判断。在此方面没有捷径可走，只能依靠项目管理者的主观判断和之前项目获得的经验。风险评估（可能性、严重性）的结果一般不是精确的数字，而是一个范围：

- 1. 对风险出现的可能性进行评估，可能的结果有风险出现的可能性非常低（<10%）、低（10%~25%）、中等（25%~50%）、高（50%~75%）或非常高（>75%）。
- 2. 对风险的严重性进行评估，可能的结果有灾难性的（严重威胁项目的存活）、严重的（可能引起很大的延迟）、可以容忍的（可能引起小的延迟）和可以忽略的。

风险分析过程结束后，应该根据风险严重程度大小按顺序制成表格。图 22-4 是对图 22-3 中已识别的风险进行分析，得出结果后做成的表格。很显然，这里对可能性和严重性的评估有些武断。为了做好这项评估，项目经理需要根据来自项目、过程、开发团队和机构状况等有关更详细的信息进行评估。

风 险	出现的可能性	后 果
开发机构的财务出现问题，必须削减项目预算	小	灾难性
招聘不到符合项目技术要求的职员	大	灾难性
在项目的非常时期，关键性职员生病	中等	严重
要复用的软件组件有缺陷，限制了项目的功能	中等	严重
需求发生变化，主体设计要返工	中等	严重
开发机构重新调整，由新的管理层负责该项目	大	严重
系统使用的数据库的处理速度不够快	中等	严重
低估了软件开发所需要的时间	大	严重
CASE 工具不能被集成	大	可容忍
客户不了解需求变更对项目造成的影响	中等	可容忍
职员所需的培训跟不上	中等	可容忍
低估了缺陷的修补率	中等	可容忍
低估了软件的规模	大	可容忍
CASE 工具产生的编码效率低	中等	可以忽略

图 22-4 风险类型和实例

当然，随着有关风险的可用信息的增多和风险管理计划的实施，一项风险出现的可能性和对这一风险的影响后果的评估都可能改变。因此，这个表格必须在风险过程的每个迭代期间得到更新。

风险一经分析和排序，下一步就该判断哪些风险是最重要的，这是在项目期间必须考虑的。做出以上判断必须综合考虑风险出现的可能性大小和该风险的影响后果。一般而言，所有灾难性的风险都是必须考虑的，所有出现的可能性超过中等、影响严重的风险同样要给予认真对待。

Boehm (1988) 曾建议识别和监控“前 10 位”风险，我认为这个数字有点太过绝对化，必须根据项目自身的情况确定要进行监控的风险的数量。可能是 5 个，也可能是 15 个。但是，选出的接受监控的风险的数目应该便于管理。所管理的风险数量太大时需要收集十分可观的信息。在图 22-4 已识别的风险中，考虑具有灾难性或严重后果的总共 8 种风险已经足够了（见图 22-5）。

22.1.3 风险规划

在风险规划过程中，项目管理者要考虑已经识别出的每一个重大风险，并确定处理这些风险的策略。对于每个风险来说，必须思考一旦某个风险发生时所需要采取的行动，使其对项目的干扰最小化。同时，应该考虑在监控项目时需要收集哪些信息，用于预测可能发生的问题。制定风险规划计划同样没有捷径可走，同样要依靠项目管理者判断和自身经验。

图 22-5 给出了处理图 22-4 中重大风险（严重的，或不能容忍的）的可能的策略。这些策略分为以下 3 类：

- 1. **规避策略** 采用这些策略就会降低风险出现的可能性。图 22-5 中处理有缺陷的组件策略就是一个例子。
- 2. **最小化策略** 采用这些策略就会减小风险的影响。图 22-5 中解决职员生病问题的策略就属于这一种。

3. 应急计划 采用这些策略, 就算最坏的事情发生, 我们也是有备而来, 有适当的对策处理它。图 22-5 中应对机构的财务问题的策略就属于这一类。

风 险	策 略
机构的财务问题	拟一份简短的报告, 提交高级管理层, 说明这个项目将对业务目标有重大贡献
职员招聘问题	告诉客户项目潜在的困难和延迟的可能性, 检查要买进的组件
职员生病问题	重新对团队进行组织, 使更多工作有重叠, 员工可以了解他人的工作
有缺陷的组件	用买进的可靠性稳定的组件更换有潜在缺陷的组件
需求变更	导出可追溯信息来评估需求变更带来的影响, 把隐藏在设计中的信息扩大化
机构调整	拟一份简短的报告, 提交高级管理层, 说明这个项目将对业务目标有重大贡献
数据库的性能	研究一下购买高性能数据库的可能性
低估开发时间	对要买进的组件、程序生成器的效用进行检查

图 22-5 风险管理策略

读者可以看到此处与在要求极高的系统中为确保可依赖性、信息安全性和安全性所采取策略的相似之处, 都必须避免、容忍失败以及从失败中恢复。显然, 最好是使用规避风险的策略。如果这办不到的话, 就采取降低那些会导致严重后果的风险的发生概率的策略 (最小化策略)。最后, 必须有成熟的应急策略以应对这些可能出现的风险, 以此降低在项目或产品上的总的风险影响。

22.1.4 风险监控

风险监控就是检查之前对产品、过程以及业务风险的假设是否改变的过程。必须要对每一个识别的风险定期进行评估, 从而确定风险出现的可能性是变大了还是变小了, 风险的影响后果是否有所改变。为了达到这个目的, 必须关注能提供有关风险可能性及其影响后果信息的其他因素, 比如需求变更的数量。它能给出风险概率和风险影响的线索。显然, 具体有哪些因素取决于风险的类型。图 22-6 举出上述因素的一些例子, 可能会对评估这些风险类型有帮助。

风 险 类 型	潜在的指征
技术	硬件或支持软件延迟交付, 暴露出来许多技术问题
人员	员工士气低迷, 团队成员的关系不协调, 工作分配不当
机构	机构内说三道四, 缺乏资深管理
工具	团队成员不愿使用工具, 抱怨 CASE 工具, 需要更强大的工作站
需求	很多需求变更请求和客户怨言
估算	跟不上双方协商的进度, 无法除掉暴露出来的缺陷

图 22-6 风险因素

风险监控应该是一个持续不断的过程, 在每一次对风险管理进行评审时, 每一个重大风险都应该单独评审并在会上进行讨论。项目管理者应该判断风险出现的可能性是变大还是变小了以及风险的严重性和后果是否也发生了变化。

22.2 人员管理

对于一个软件机构来说, 人是这个机构中最重要的资产。招募和留住好的员工是需要巨大

开支的，软件管理者最终决定着机构能确保从对员工的投资中获取最大的回报。成功的公司和经济实体尊重它们的员工，并且在对他们的任务分配上体现他们的技能和经验的价值，以此达到上述目标。

软件项目的管理者对影响软件开发工作的技术问题的理解是很重要的。然而不幸的是，好的软件工程师并不一定是好的人力资源管理者。软件工程师通常拥有娴熟的技术技巧，但是却缺乏能够激励和领导项目开发团队的软技术。作为项目管理者，应该意识到人员管理的潜在问题，并且应该尝试不断提高自身在人员管理方面的技巧。

在人员管理方面有 4 个关键要素：

- 1. **一致性** 对项目组的每个人应该同等对待。虽然没有人会期望每个人的报酬都相同，但是却能感觉自己机构的贡献遭到低估。
- 2. **尊重** 不同的人有着不同的技能，管理者应尊重这些差异。团队的所有成员都应有机会做出贡献。当然在某些情况下你会发现某个人根本不适合在这个团队，所以也不能够继续下去，但重要的是不要匆忙得出这样的结论。
- 3. **包容** 当人们觉得其他人能够倾听并能采纳自己的建议时，他们能更有效地工作。很重要的一点是创造一种工作环境，在这种工作环境下，所有的见解，甚至是最底层员工的意见都会得到考虑。
- 4. **诚实** 作为一个管理者，必须对项目组中好的情况和不好的情况保持诚实的态度。应当诚实地对待自己的技术知识水平，并且在必要的时候乐于服从更博学的员工。如果掩盖存在的问题和自己的无知，最终将被发现并且失去团队的尊敬。

在作者看来，人员管理只能从实际经验中学，而不是从书本中学。本节和下一节（团队协作）的目的在于介绍能够影响软件项目管理的一些最重要的人员和团队管理问题。希望这些资料能让管理人员意识到当管理一群技术上有天赋的人所组成的团队时可能遇到的一些问题。

工作动力

项目管理者需要激励一起工作的员工，使他们能尽心尽力。激励意味着在协调工作和提供良好工作环境方面能使员工尽可能更有效率地工作。如果员工没有被激励起来，他们将失去对正在从事的工作的兴趣，工作进度会很缓慢，会更容易犯错误，并且不能对团队或机构长远的目标有所贡献。

为了达到激励员工的目的，需要理解什么才能激励员工。Maslow（1954）建议通过满足人的需求来激励他们，这些需求被划分成一系列层次，如图 22-7 所示。这里较低的层次表示基本的需求，如对食物、睡眠等的需求，还有在某一环境下对安全的需求。社会需求是指对某一社会群体的认同感。受尊重需求是指得到其他人的尊重，自我实现的需求是指个人发展。人类首先要满足低层次的需求，如饥饿，然后是更抽象更高层次的需求。

软件开发机构中的员工根本没有饥渴问题，一般也不会感受到环境对身体的威胁。因此，确保社会需求、受尊重需求和自我实现需求的满足，从管理的观点来看是最有意义的：

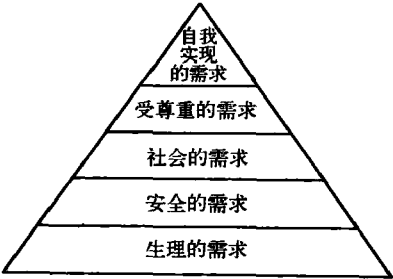


图 22-7 人的需求层次

- 1. 满足员工的社会需求，就是给员工提供与同事交往的时间和场所。当所有开发团队成员在一个地方工作时，这种需求相对比较容易满足，但越来越多的团队成员并不在同一个建筑物中

工作，甚至不在同一个城市或一个国家内。他们可能同时在为多家公司服务，还可能绝大多数时间里是在家工作。

社交网络系统和电话会议可使交流变得更便捷，但以作者的经验来看，电子系统只有当人们互相认识的情况下使用才最有效。所以，在项目的初期应当安排定期的面对面的会议，以使人们能够直接与团队的其他成员进行交流。通过这种直接交流，人们成为这个团队的一部分，并理解团队的目标和重点。

2. 为了满足员工的受尊重需要，应该让员工们感觉到他们在开发机构中很受尊重。对员工做出的成绩给予认同就是一种简便有效的方式。显然，也必须让员工们感觉到为他们所支付的报酬能够反映出他们的能力和经验的价值。

3. 最后，为了满足员工的自我实现需要，应该让员工对自己的工作负起责任，分配给他们比较难的（但不是不能完成的）任务，并给他们提供培训计划以提高他们的技能。因为员工喜欢学习新的知识和技能，所以培训是一项重要的激励方式。

图 22-8 说明了一个管理者通常不得不面对的关于工作动力的问题。在这个例子中，一个非常有能力的小组成员对工作和小组都失去了兴趣。于是，其工作的质量开始下降，变得不可接受。这种情况必须尽快得到处理。如果不能解决这一问题，小组的其他成员将会开始不满，觉得受到不公平的待遇。

案例分析：动力

Alice 是个软件项目经理，她所在的公司主要研发报警系统。公司最近试图渗透到发展很快的老年人和残疾人市场，开发帮助这个群体独立生活的技术产品。Alice 被指定领导一个 6 人研发团队开始这个方面的项目，不再继续搞她所熟悉的报警技术。

Alice 的辅助技术项目进行得很顺利。在组内建立了良好的工作关系，一些具有创新的想法得到采纳。公司决定采用链接到网络的数字电视建立一个点对点的消息系统。然而，几个月之后，Alice 发现硬件设计专家 Dorothy 开始上班迟到，工作效率下降，而且没有和其他成员进行交流。

Alice 和小组的其他成员进行了非正式的谈话，试图了解 Dorothy 的个人环境是否发生了改变，是否影响到了她的工作。但是他们都不太清楚，所以 Alice 决定直接和 Dorothy 谈谈，了解其问题。

在几次否认之后，Dorothy 承认她对工作失去了兴趣。她希望能发展和运用她的硬件接口经验。但由于产品方向已经确定，在这方面她只有很少的机会。基本上她只是同其他团队成员一起作为 C 语言程序员在工作。

在她承认这项工作是一种挑战时，也同时明白在此她不能拓展擅长的接口技术。她很担心当进行完这个项目以后，将会很难再去找到一个包含有硬件接口的工作。因为她不想让团队知道她正在考虑下一份工作而影响到别人的工作热情，所以决定最好与团队中尽可能少的人谈论这件事。

图 22-8 个人激励

在这个例子中，Alice 试着去弄清楚 Dorothy 的个人情况是否有问题。私人问题通常会影响到工作热情，因为人们不能专心于他们的工作。必须给他们时间并支持他们去解决自己的问题，同时也必须让他们清楚地认识到需要对雇主负责。

实际上 Dorothy 所面临的动力问题在项目开发与人们所预期的方向不一致时经常会发生。人们期望去做一种类型的工作，但实际上可能最终却干着完全不同的事情。当人们想以某种方式扩展其技术能力，而在所做的项目中却无法达成时，这会成为一个问题。在这个例子中，可能要决定让这个团队成员离开到别的地方寻找机会。而这里 Alice 试着去说服 Dorothy 拓展经历是职业生涯中一个正确的选择。她给了 Dorothy 更多的设计自主权，并组织了一些软件工程方面的培训课程，以使 Dorothy 在结束当前的项目后能有更多的机会。

Maslow 的动力模型在一定程度上是很有帮助的，但是此模型存在的问题在于他对于动力的

理解只是从人本身这个角度来看。他没有充分考虑到这样的事实：人们认为自己是一个机构、一个专业群体以及某种文化中的一部分。这并不仅仅是一个满足人们社会需要的问题，人们也可以通过帮助一个团体达成共同的目标而受到激励。



人的能力成熟度模型

人的能力成熟度模型（P-CMM）是一个评估框架，是评估机构在管理员工自身发展方面的水平的。它强调在人的管理方面的最佳实践，以及提供给机构改善它们的人的管理过程的基础。

[http://www. SoftwareEngineering-9. com/Web/Management/P-CMM. html](http://www.SoftwareEngineering-9.com/Web/Management/P-CMM.html)

作为一个有凝聚力的群体的成员，对多数人而言就是一个很大的动力。人们完成任务后常常乐意继续工作，工作的动力来自于他们的同事及其所做的工作。因此，在考虑激励每个人的同时，也必须考虑如何将团队作为一个整体进行激励以达到公司的目标。下一节将讨论团队管理的问题。

人格类型也会对激励产生影响。Bass 和 Dunteman（1963）将职业人士分为 3 种类型：

1. 面向任务型，这类专业人员的动力来自于他们所从事的工作。在软件工程中，他们是这样的人：软件开发智力上的挑战激发了他们的工作热情。
2. 面向自我型，这类人的动力主要来自于个人成功和得到认可。他们更乐于把软件开发视为达到自己目标的手段。这并不意味着这些人自私，只关注自己的事情。他们通常有更长远的目标，比如得到升迁，他们希望在工作中获得成功以帮助实现自己的这些目标。
3. 面向交互型，这类人的动力来自于同事们的存在和协作。随着软件开发日益以用户为中心，面向交互的人们也越来越积极地参与到软件工程中来。

面向交互型人员通常喜欢小组作业，而面向任务型和面向自我型人员则通常喜欢独自工作。女人比男人更易成为面向交互型人员，她们通常是更有效的交流者。在图 22.10 节中讨论了团队中这些不同的特性。

每个个体的工作动力由各种动力因素组成，但是在任一时刻总是只有一种动力居于支配地位。然而每个人都可能发生改变。举个例子，如果技术人员对所支付的报酬不满意，那么他很可能变成面向自我型，把个人利益置于技术乐趣之上。如果团队协调得特别好，面向自我型的人也可能变成面向交互型的。

22.3 团队协作

多数专业软件都是由一个项目团队开发完成的，这些团队的规模从两人到几百人不等。然而，让大型团队中的每个人都能有效地参与、解决一个问题，很显然是不可能的，通常要把这些大型团队分成小的项目小组，每个项目小组各负责整个系统的一个子项目。一般原则是，软件工程项目小组的成员通常不应该超过 10 人。分成小型的项目小组可以减少沟通中的问题。人人都互相认识，整个小组可以坐在一起开会讨论项目和当前正在开发的软件。

组成一个高效的项目小组是一项至关重要的管理任务。这个小组应该在技术、经验和个性方面整体均衡。成功的小组并不只是各种技能均衡的个体的简单组合。好的小组具有一种凝聚力和团队精神，使得所有的成员既为自己的个人目标奋斗同时又为小组的成功而奋斗。

在有凝聚力的小组中，成员认为集体比个人重要。领导得当，凝聚力强的小组的成员都忠于小组，他们认同集体目标和其他成员。他们试图保护集体为一个整体，免受外来的干扰。这就使

得小组足够健壮，有能力解决各种问题和突发状况。

小组的凝聚力带来的好处如下：

1. **能够建立起小组自己的质量标准** 因为这个标准的建立是经过小组一致同意的，与外部强加给小组的标准相比，更容易被大家遵守。

2. **成员互相学习，互相帮助** 小组中的成员之间互相学习。鼓励互相学习可以消除由于相互不了解而引起的隔阂。

3. **知识分享** 一旦有成员离开小组也能够保持工作的连续性。小组的其他成员可以接手关键的任务，保证不会过度影响项目。

4. **鼓励重构以及不断改善** 不论是谁最先开始创建的这个设计或程序，小组成员集体工作交付高质量的产品并解决出现的问题。

优秀的项目管理者应该鼓励小组凝聚力。他们可以为小组成员及其家庭组织一些社会活动，可以通过给小组命名确立小组的特性和定位，尝试着建立小组的认同感，还可以开展有鲜明小组特色的小组建设活动，如运动和游戏。

增强小组凝聚力的最有效的方式是把组员当做自己人。必须认为小组成员是负责任的、可信赖的，保障小组成员的知情权。管理者们常常觉得某些信息不能让小组所有的成员都知道，这样难免会产生相互之间的不信任。坦诚的信息交流是一种简便而又有效的方式，可以使小组成员感觉他们是小组的一部分。

在图 22-9 中的案例分析是一个例子。Alice 安排非正式的例会通知其他的小组成员将要做什么。她重视让成员们提出来自自己小组经验的新想法，从而参与到产品开发中来。“放松日”也是提高凝聚力的一个好方法。当人们互相帮助学习新技能时他们也一起得到了放松。

案例分析：团队精神

Alice 作为一个有经验的项目经理，了解建立一个有凝聚力的小组的重要性。当开发一个新产品时，她让负责产品描述的小组和产品设计的小组的所有成员都参与到与各小组中资深者一起对可能用到的技术的探讨活动中来，她利用这样的机会使得项目团队中的每个成员都能够关心产品的描述和设计。她还鼓励老成员介绍新成员与其他小组中的成员认识。

Alice 每月都安排午餐会，这是项目组所有成员相互认识并谈论彼此所关心的问题的机会。在餐会上 Alice 告诉小组成员她所知道的公司的新闻、制度、策略等。然后每个成员简要总结他们已经完成和正在进行的工作，然后是小组讨论一般性话题，例如从年长亲戚那里获得的新产品构想。

每隔几个月，Alice 组织一次小组的“放松日”，用两天时间对小组进行“技术提升”。每个小组成员准备一项相关技术的最新内容并介绍给小组中其他成员。到一个较远的地方选择一个高级宾馆度假，给出充足的时间进行充分的技术讨论和成员彼此间相互沟通。

图 22-9 小组凝聚力

在一定程度上，一个小组能否有效率地工作取决于项目本身以及承担该项目的机构。如果这个机构不断重组，工作不安全，处于一片混乱之中，团队成员很难专注于项目开发。然而，除项目和机构问题之外，还有 3 个影响团队工作的普遍因素：

1. **小组的人员** 项目小组需要不同类型的人员。因为软件开发包括各种活动：和客户谈判，编码，测试，编写文档等。

2. **团队的组织** 应该这样组织团队，使得小组成员都能尽其所能，所承担的各项任务都能按时完成。

3. **技术上和管理上的交流** 小组成员之间、软件开发团队和其他项目信息持有者之间的良好沟通是必不可少的。

正如所有的管理问题一样，获得合适的团队并不能保证项目成功。还会有太多的地方可能出现问題，如业务变更和业务环境改变。然而，不注意小组构成、组织以及交流的话，项目出现问题的可能性就大大增加了。

22.3.1 成员挑选

管理者或团队领导者的任务是创建一个有凝聚力的小组并将其很好地组织起来，使其能一起有效地工作。这包括两方面：创建一个在技术技能和人格个性之间平衡的小组；组织成员一起有效地工作。有时，员工是从组织外面雇佣的；然而更常见的是，软件工程小组是由从其他项目中有经验的现有职员中挑选来的。然而，事实情况是管理者很少有机会挑选小组成员。他们经常不得不使用公司里能找到的人员，即使这些人并不是这项工作的理想人员。

如 22.2.1 节所讨论的，许多软件工程人员的动力主要来自于他们的工作。因而软件开发小组常常都是由一些对技术问题解决有独到见解的人员组成的。有无独到见解可以从定期报告的问题中得到证实，这些问题包括接口标准被忽视、将系统用他们自己的编码重新设计过、不必要的系统修饰等。

由性格互补的成员组成的小组比仅仅根据技术能力选择成员的小组更有效率。以工作为动力的人很可能在技术上是最出色的。面向自我的人则可能善于推动整个工作的完成。面向交互的人有利于小组内部的交流沟通。作者认为一个小组拥有面向交互型的人员特别重要。这种类型的人喜欢和别人交谈，可以较早地发觉小组中的紧张与不和谐状态，使得紧张与不和谐状态不致对小组造成严重影响。

图 22-10 中所示的案例中，表现了项目经理 Alice 如何去组建一个个性和谐的小组。这个特别的小组中有着面向交互和面向任务型两种人的良好组合，但是在图 22-8 中讲到了 Dorothy 面向自我的个性可能会导致一些问题，因为她干的并不是她希望的工作。Fred 作为领域专家的兼职角色也可能是个问题。他太过专注于技术的挑战，因而可能不能与其他的小组成员进行良好的互动。他不能始终作为小组的一部分，这意味着他可能不能很好地与团队的目标相一致。

案例分析：小组构成

在创建辅助技术开发小组的过程中，Alice 认识到选择个性和谐成员的重要性。面试的时候，她试着判定他们是面向任务型、面向自我型还是面向交互型的。她觉得自己主要是面向自我型的，因为她觉得这个项目是使得她能获得上层注意和得到提升的一个途径。因此她寻找一个或两个面向交互型的人并期望能找到一些面向任务型的人来建立小组。她最后的看法是：

Alice——面向自我型
 Brian——面向任务型
 Bob——面向任务型
 Carol——面向交互型
 Dorothy——面向自我型
 Ed——面向交互型
 Fred——面向任务型

图 22-10 小组构成

有的时候不可能挑选个性互补的成员组成一个小组。在这种情况下，项目管理者必须控制小组，不能让小组成员把个人目标凌驾于开发机构和小组的目标之上。如果所有的成员都参与到项目的各个阶段，这种控制比较容易实现。当项目小组成员在接受指示的时候不知道他们的任务在整个项目中的地位，他们很有可能会按照自己的意志行事。

举个例子，一个工程人员要为一个程序做编码设计，并注意到了一些可能的设计改善。如果在实现这些改善时，工程人员没有理解设计的初衷，任何改变，虽然初衷是好的，都会对系统的其他部分产生不利影响。如果整个小组从一开始就参与这项设计，他们就会理解为什么会做出这样的设计决定。他们就会认同这些决定，而不再反对它们。



雇用好的员工

项目管理者通常负责挑选机构中的员工到他们的软件工程团队来。在此过程中获得可能的最好的员工是非常重要的，因为坏的选择决定会是项目的一个严重风险。

影响员工挑选的一些关键因素包括：教育背景 and 培训，应用领域和技术经验，沟通能力，适应性以及问题解决能力。

<http://www.SoftwareEngineering-9.com/Web/Management/Selection.html>

22.3.2 小组的结构

小组的构成方式影响着小组的决定、信息交换的方式以及开发小组和小组外的项目信息持有者间的交流。项目管理者关心的重要组织问题如下：

1. 项目管理者应该是小组的技术领导人员吗？技术领导人员或者系统架构师负责项目开发过程中的重大技术决定。有些情况下，项目管理者有足够的技术和经验但当这个角色。然而，对于大型项目，最好指派一位资深的工程师但当项目架构师，并负责技术领导任务。

2. 谁将参与做出重大技术决定，以及如何做出？应该让系统架构师、项目管理者还是让更多的小组成员一起做出？

3. 如何与小组外的信息持有者以及高层管理交流沟通？在很多情况下，项目管理者将会在系统架构师（如果有的话）的辅助下负责这些沟通。另一个可以选择的组织模式是指派一个有着良好沟通能力的人担当此角色，专门负责外部联络。

4. 如何将分散的人员整合到一组？很普遍的情况是小组包括不同机构的成员，有的成员在共同的办公室办公也有人在家办公。在小组决策过程中应该将此考虑在内。

5. 小组怎么共享知识？小组的组织方式影响信息的共享，比起其他方式，某些组织方式更有利于共享知识。然而，也应该避免共享过多的信息，以免超过成员承受的范围，过多的信息会使成员在工作中分心。

小型的程序设计小组的组织结构通常是非正规的。小组领导和其他小组成员一起参与软件开发。在非正规结构的小组中，完成哪些工作要经过小组集体讨论，任务的分派按照个人能力和经验进行。高级的系统设计由小组的资深成员完成，低级的设计则由承担具体任务的小组成员负责。

极限编程小组（Beck, 2000）总是非正式小组。XP（极限编程）热衷者认为正式的组织结构阻止信息交流。在这种方式中，许多通常被认为应由管理层做出的决策，如对项目调度的决策，都交给了小组成员。程序员一起写程序代码，集体对开发的程序负责任。

如果小组的大多数成员都既有经验又有能力，非正规结构的小组会做得非常成功。小组的运作实行民主、集体决策。这样增强了小组的凝聚力和实力。然而，如果小组的大部分成员经验不足，或者能力不足，非正规结构会阻碍小组的发展。没有一定的权威来指导工作的进行，会使得小组成员间缺乏协调，并有可能最终导致项目失败。

层次化小组是将小组看成一种层次结构，小组领导人员处于最上层。领导者比起其他组员

有更大的权限，所以可以指导工作。组织层次结构清晰，结构上层做出决定，结构下层执行决定。在这种结构中，交流主要是高层人员下达的指示，从下层到上层的交流相对来说就很少了。

当一个深刻理解的问题能很轻易地分解成数个子问题，这些子问题能够在这个层次结构中的不同位置找到解决方案，这种结构是很有效的。在上述情况中，层次中交流的要求相对较少。然而，在软件工程中这样的情况是相当罕见的，原因如下：

1. 软件的变更通常会引起系统若干部分的变更，因此需要层次结构中各个等级的人员一起讨论磋商。

2. 软件技术飞速发展，年轻职员往往比有经验的老职员对技术懂得更多。自上而下的交流方式意味着项目管理者找不到机会使用新技术。而且由于在开发中使用在他们看来是过时的技术，越来越多的新员工会产生挫败感。

民主的和层次的小组组织方式没有认识到组员之间的技术能力存在巨大的差异。最好的程序员的生产力能比最糟的高出 25 倍。因此需要为最好的人尽可能地提供支持，以最有效的方式使用他们。一个早期的提供这种支持的组织模式是“首席程序员团队”。

为了使技术熟练的程序员人尽其才，Baker (Baker, 1972) 和其他的学者 (Aron, 1974; Brooks, 1975) 提出团队应该单独设立技术熟练的首席程序员。设首席程序员的团队的一个基本原则是首席程序员应该是熟练的、有经验的成员，他们应该对软件开发全过程负责。不应该让他们处理日常事务，他们的工作应该得到技术上和行政上的支持。他们应该专注于要开发的软件，而不要在这些无关的会议中浪费时间。

但是在作者看来，采用首席程序员的团队结构过于依赖首席程序员和他们的助理。其他的团队成员由于不能被赋予足够的权利会变得消极，因为他们觉得自己的技能没有充分发挥。出现问题的时候，他们处理问题所需信息不足，他们也没有机会参加决策过程。这种组织结构会带来巨大的风险，而且这些风险可能超过它带来的任何好处。



物理工作环境

人们所处的工作环境既影响小组沟通也影响每个人的工作效率。独立工作间对从事复杂的技术工作的人集中注意力特别有帮助，这样技术人员就会最大限度地受到干扰。然而，共享空间特别有助于沟通。设计良好的环境应该综合考虑到这两方面的需要。

<http://www.SoftwareEngineering-9.com/Web/Management/workspace.html>

22.3.3 小组的沟通

组员相互之间以及和其他项目信息持有者之间的准确高效的沟通是绝对必要的。组员之间需要交流各自工作的情况，做出的设计决策，以及对原先设计决策进行修改。组员必须解决其他信息持有者所提出来的问题，并告之在系统方面、小组方面以及在交付计划方面等的变更。良好的交流有助于增强小组的凝聚力。组员开始理解小组中其他人员的动机、长处以及弱点。

影响沟通有效性和效率的主要因素有：

1. **小组规模** 随着小组规模的扩大，要保证所有的成员都能彼此有效交流就变得很困难了。单向交流链的数目是 $n \times (n - 1)$ ，其中 n 代表小组的人数，可以看出，对于一个有 8 个成员的小组，有 56 条可能的交流路径。有些成员之间很可能没有实际交流。小组成员之间职位上的差异意味着沟通常常是单向的。管理者和经验丰富的工程师在与经验较少的成员进行交流时，容易处于支配地位，谈话的对象常常不愿主动开始交谈。

2. 小组结构 在一个没有正规组织结构的小组中,员工的沟通比有正规层次化结构的小组中员工的沟通更为有效。在有层级结构的小组中,沟通是按照层级结构进行的。处于同一层级的人可能彼此不进行交流。这是含有几个开发小组的大型项目中的一个突出问题。如果开发不同子系统的人员只与他们的管理者进行沟通,通常会造成项目的延期和理解上的错误。

3. 小组构成 如果小组中人太多并且个性类型相同(在22.2节讨论过),这样可能容易发生冲突,沟通也不能正常进行。由混合性别组成的小组(Marshall和Heslin,1975)中的沟通要比由单一性别组成的小组容易进行。女性比男性更容易成为面向交互型人才,小组中的女性成员可作为小组沟通的控制员和协调员。

4. 小组的物理工作环境 工作场所的结构是推动或阻碍沟通的主要因素。可以在本书的网站上找到更多关于这个问题的信息。

5. 可利用的交流渠道 交流的形式有很多——面对面的方式、发电子邮件、正式文件、电话以及Web 2.0技术比如社交网络和Wikis。项目组之间分布在各处的情况越来越常见,组员都在远地工作,需要广泛采用能使交流变得便捷的技术。

对项目管理者来说,截止日期通常都很紧,所以他们更愿意使用不太费时的交流渠道。他们依靠会议以及正式文件向项目职员和信息持有者传达信息。虽然从一个项目管理者的角度来看这是有效率的交流方式。但通常却没有效果。我们可以发现有很多原因使得人们不去参加会议,所以他们听不到报告。冗长的文件几乎不会被阅读,因为阅读者不知道文件是否和自己相关。当同样的文件产生了几个版本的时候,读者会很难找到其中的变化。

当沟通是双向的时候,有效的沟通就达到了。有关的人员讨论问题和咨询,并且建立起对提议和问题的共同的理解。可以通过会议达到目的,虽然会议通常被强势的人员所控制。有时候安排会议也是不实际的。越来越多的项目团队中包括在外地工作的人员,这使得举行会议更加困难。

为了解决这些问题,可以使用Web技术比如Wikis和博客支持信息交流。Wikis支持合作创造和编辑文档,博客支持主题为线索的讨论,组员可以发出问题和评论。无论在哪,Wikis和博客允许项目成员和外部信息持有者交换信息。它们能帮助管理信息,分辨讨论的类型,而这些使用电子邮件的时候经常把人搞糊涂。当然,也可以使用即时通信工具和电话会议处理需要讨论的问题。这是很容易安排的。

要点

- 软件工程项目要想按进度、按预算进行开发,完善的软件项目管理是必要的。
- 软件项目管理与其他的工程管理有明显区别。软件产品是无形的。软件项目可能很独特或者有创新,这样就没有实在的经验以供借鉴。软件过程没有传统工程过程那么成熟。
- 风险管理现在被看做最重要的项目管理任务之一。
- 风险管理包括识别并评估重大的项目风险,从而判断这些风险发生的可能性大小及其后果的严重程度。对很有可能发生并有潜在严重性的风险,应该制定有关规避、管理和解决可能发生的风险的计划,包括对风险发生的分析和当风险发生时的应对措施。
- 对一个人的激励因素会包括与其他人的交互作用、得到管理层和同行的赏识,以及得到个人发展的机会等。
- 软件开发小组不宜太大且要有凝聚力。影响小组效率的关键因素是小组中的人员、小组组织的方式以及组员之间的沟通。
- 一个小组内部的沟通受许多因素的影响,如小组成员的职位、小组的规模、小组的性别组成、小组成员的个性以及可用的沟通渠道。

进一步阅读材料

《The Mythical Man Month (Anniversary Edition)》自 20 世纪 60 年代以来，软件管理的难题一直就没有变化，这是软件管理方面的最好的书之一。该书有趣、易懂，描述了最早的大型软件项目之一的 IBM os/360 操作系统的管理活动。在第 2 版中还收入了 Brooks (F. P. Brooks, 1975, Addison-Wesley) 的其他一些经典的论文 (F. P. Brooks, 1995, Addison-Wesley)。

《Software Project Survival Guide》这是一本十分注重实效的软件管理著作，包含很多对有着软件工程背景的项目管理者来说很好的具体建议，很容易阅读和理解 (S. McConnell, 1988, Microsoft Press)。

《Peopleware: Productive Projects and Teams, 2nd ed》这是一本经典著作的新版本，强调了软件项目管理中人文关怀的重要性。这本书容易阅读，是少有的认可人的工作场所的重要性的书之一。作者极力推荐此书 (T. DeMarco and T. Lister, 1999, Dorset House)。

《Waltzing with Bears: Managing Risk on Software Projects》这是一本实践性很强且易读的书，主要介绍风险和风险管理 (T. DeMarco and T. Lister, 2003, Dorset House)。

练习

- 22.1 解释软件系统的无形性为什么给软件项目管理提出了特殊的难题。
- 22.2 解释最好的程序设计者为什么不一定能成为最好的软件管理者。22.1 节中列出的管理活动可以帮助你回答这个问题。
- 22.3 根据文献中已报告的项目问题的实例，列出这些失败的程序设计项目中存在的管理难点和错误（可参考 Brooks 的书，在“进一步阅读材料”中已经推荐过）。
- 22.4 除了图 22-1 中所列的风险，识别可能在软件项目中出现的其他 6 种风险。
- 22.5 价格锁定的合同，即承包人以某个确定价格投标一系统开发，是一种将项目风险从委托人转移给承包人的做法。如果出现问题，需要由承包人承担。分析一下为什么这种合同容易增加产品风险的可能性。
- 22.6 为什么让小组所有成员都知悉项目的进展情况和技术上的决策，能够增强小组的凝聚力？
- 22.7 在极限编程团队中许多管理决策权被下放到团队成员自己手中，你认为这会带来哪些问题？
- 22.8 编写一个类似本书风格的案例研究，证明项目组中沟通的重要性。假定一些组员在外地工作，不可能在短时间内召集在一起。
- 22.9 你的上司要求你在某一时间之前交付软件产品，而据你了解只有让你的项目组无偿地加班加点，才能赶上进度，而项目组所有的成员都有孩子需要照顾。你应该接受上司的要求，还是应该说服你的项目组成员加班，讨论一下。做出你的决定要考虑哪些重要因素？
- 22.10 作为一个程序设计员，你被提升为项目经理，但是你感觉做具体的技术工作比担任项目经理做出的贡献要大。讨论一下你是否应该接受这一提升。

参考书目

- Aron, J. D. (1974). *The Program Development Process*. Reading, Mass.: Addison-Wesley.
- Baker, F. T. (1972). 'Chief Programmer Team Management of Production Programming'. *IBM Systems J.*, 11 (1), 56-73.
- Bass, B. M. and Duntzman, G. (1963). 'Behaviour in groups as a function of self, interaction and task orientation'. *J. Abnorm. Soc. Psychology.*, 66 (4), 19-28.
- Beck, K. (2000). *extreme Programming Explained*. Reading, Mass.: Addison-Wesley.

Boehm, B. W. (1988). 'A Spiral Model of Software Development and Enhancement'. *IEEE Computer*, 21 (5), 61-72.

Brooks, F. P. (1975). *The Mythical Man Month*. Reading, Mass.: Addison-Wesley.

Hall, E. (1998). *Managing Risk: Methods for Software Systems Development*. Reading, Mass.: Addison-Wesley.

Marshall, J. E. and Heslin, R. (1975). 'Boys and Girls Together. Sexual composition and the effect of density on group size and cohesiveness'. *J. of Personality and Social Psychology*, 35 (5), 952-61.

Maslow, A. A. (1954). *Motivation and Personality*. New York: Harper and Row.

Ould, M. (1999). *Managing Software Quality and Business Risk*. Chichester: John Wiley & Sons.

项目规划

目标

本章的目的是概要介绍软件项目规划、项目进度安排以及成本估算，通过阅读本章，你将了解以下内容：

- 理解软件成本计算的基本原理和软件报价不能直接与它的开发成本相关联的原因；
- 知道在计划驱动的开发过程中的项目计划应该包含哪些部分；
- 理解项目进度安排包含的内容以及如何使用条形图制作项目进度安排；
- 了解在极限编程中用来支持项目规划的“规划游戏”；
- 理解如何利用 COCOMO II 模型进行算法成本估计。

项目规划是软件项目管理者最重要的工作之一。项目管理者必须将工作分解开来并分配给团队成员，必须预见可能出现哪些问题，并且准备好相应的试探性的解决办法去应对这些问题。项目计划是在项目的开始建立的，是用于向项目开发团队和客户说明工作如何开展，以及帮助估计项目进展的。

项目规划发生在项目生存周期的以下 3 个阶段：

1. 投标提案阶段，即为争取一个开发软件系统开发合同的投标阶段。在这个阶段，需要做出计划以帮助管理者判断是否有完成这项工作所需的资源，并计算出向客户开出的价格。

2. 项目开始阶段，此时需要做出如下规划：确定参加此项工作的人员；将工作分解成数个子项目；在公司中如何分配资源等。此时，已经掌握了比投标阶段更多的信息，因此可以进一步完善最初的工作量估计。

3. 贯穿于项目过程中。定期地根据获得的经验和项目进展的监督信息修改计划。项目管理者对开发的项目和开发团队的能力了解越多，就能越准确地估计项目的开发时间。此外，软件需求可能发生改变，这就意味着项目完结时间不得不改变，项目时间延长。对于传统开发项目，这意味着开始阶段创建的计划必须修改。但是，如果使用敏捷方法的话，计划的时段更短且是随着软件开发持续变化的。23.4 节将讨论敏捷规划。

通常，由于在投标阶段没有要开发的项目的完整需求，不可避免提案阶段的规划只是预测性的。但是必须按照客户要求拿出提案，对基于所要求的软件功能做出高水平描述。规划通常是提案要求的一部分，所以，必须做出对于开展工作可信的规划。如果能够赢得合同，通常必须重新规划项目，将提案做出后所出现的变化考虑在内。



经常费用成本

当你估计在软件项目上的工作量成本时，不能简单地用人员工资乘以投入到项目的时间。你还必须考虑所有的机构的日常开支（办公空间，管理等），这些都应该是项目收入所承担的。通过计算这些日常开支，加上项目中工程师的成本乘上一个系数，就是项目的总的成本。

<http://www.SoftwareEngineering-9.com/Web/Planning/overheadcosts.html>

在投标争取一个合同的时候，必须计算出需向客户提出的开发软件的价格。计算价格首先需要估计完成项目所需要的成本。这包括计算完成各个活动所需要的工作量，然后计算所有活动的工作量。项目管理者必须客观地计算软件成本，才能准确地预测开发软件的成本。一旦对可能的工作量有合理的估计，接下来才能计算出价格，向客户提出报价。但是正如下一节要讲的，这不只是简单地成本 + 利润，还有很多因素影响软件项目的定价。

在计算软件开发项目总成本时要分析以下3个方面：

- 工作量成本（支付给软件开发人员的费用）。
- 包括维护在内的硬件和软件费用。
- 差旅费和培训费用。

对于绝大多数项目，主要的成本是工作量成本。项目管理者必须估算完成此项工作可能需要的总成本（人月）。很显然，所掌握的用于估算的信息是有限的，所以必须首先做出最贴近的估算（往往是偏于乐观的），然后加上一个很大的应急开支（额外的时间和成本）。

对于商业系统来说，正常的情况下会使用相对便宜的商品硬件。然而，假如需要得到中间件和平台软件的授权的话，软件成本就会大大增加。如果项目是在不同地方开发，可能还需要长途出差。虽然差旅费通常只是工作量的一小部分，但是花在旅途上的时间就被浪费掉了，极大地增加了项目的工作量成本。可以使用电子会议系统和其他支持远程协作的软件减少差旅次数。节约的时间可用于更有回报的工作内容。

一旦获得开发系统的合同后，应该立刻改进该项目的框架规划，创建项目的启动计划。这个阶段管理人员需要知道更多的系统需求。然而，此时可能还没有完整的需求清单。特别是使用敏捷方法开发的话更是如此。此时的目标是创建一个能够帮助项目人员决策补充和做出预算的项目规划。将这个规划作为基础，在机构内部分配项目资源，并且帮助决定是否雇用新员工。

上述规划也应包括项目监督机制。项目管理者必须时刻追踪项目的进展，比较实际的进展和成本与原计划的进展和成本的不同。大部分的机构都有正式的监督机制，但是从监督来说，一个好的项目管理者应该能够从与员工的非正式交谈中对项目的情况建立起清晰的认识。非正式监督能够在困难出现时及时发现，从而预测到潜在的项目问题。比如，可能在与员工的日常交谈中暴露出在查找软件漏洞中的特殊问题。从而项目管理者可以立刻安排一个专家解决问题，或者决定编程绕过该问题，而不用等到员工进行进度汇报。

项目计划总是随着项目的进展而改变。要保证项目计划成为员工理解要达到的目标和软件交付的时间的一份有用的文件。因此，随着软件开发的推进，需要修改进度安排计划、成本估计以及风险估计。

如果使用敏捷方法，仍然需要项目启动计划。不论使用的是什么方法，公司仍然需要计划如何给项目分配资源。然而，这并不是一个详尽的计划，只是包括一些关于工作分解和项目日程表这些有限的信息。开发过程中，要为每个软件版本制定一个非正式的项目计划和工作量成本估计，应该让团队所有员工都参与到规划过程中来。

23.1 软件报价

原则上，对于客户来说，软件产品价格只是简单的开发成本加利润。不过，实际上项目成本和向客户提出的价格之间的关系并不总是这样简单。软件报价必须考虑到方方面面的因素，如机构因素、经济和政治因素、商业上的因素等。必须考虑的因素如图 23-1 所示。必须考虑机构上的问题，项目相关的风险，以及将要使用的合同的类型。这些都将引起价格上升或者下降。由于项目报价要考虑机构的具体情况，因而项目报价需要市场和销售人员、资深管理人员，以及项目管理者共同参与决定。

因 素	描 述
市场机遇	开发机构可能为进入一个新的软件市场而提出一个低的报价。在一个项目上的低回报可能会换来今后更大收益的机会，而且获得的经验可有助于开发新产品
成本估算的不确定性	如果机构对成本估算不太确定，它可能增加应急开支项，使得提出的报价超出了一般收益
合同条款	客户可能愿意开发者保留对源代码的版权，并在其他项目中复用。这样付出的价钱会比将软件源代码交给客户时少些
需求易变性	如果需求可能会发生改变，机构会降低它的价格以得到合同，在合同签订后，需求的改变将带来高的要价
财务状况	处于财务困难中的开发者可能会降低报价来得到一份合同。比正常情况下少赚些甚至亏一点也比没有项目好。在困难时期现金流比利润更重要

图 23-1 影响软件报价的因素

为了说明项目定价的一些问题，考虑下面的情景：

一个叫 PharmaSoft 的小软件公司，雇用 10 个软件工程师。公司刚完成了一项大工程，但是现在得到的合同仅需要 5 个开发人员就够了。但公司正在竞标一个主要的制药公司的大合同，需要在 2 年内完成 30 人年的工作。项目在至少 12 个月内不会动工，但一旦获得批准，该项目会让公司的财务状况有很大起色。

PharmaSoft 公司得到一个机会竞标一个需要 6 个人且在 10 个月内完成的项目。成本（包括项目的日常开支）估计在 120 万美金。但为了提高竞争力，PharmaSoft 公司给顾客的标价是 80 万美金，这就意味着尽管在这个合同中损失了一些收入，但它为一些在一年时间内就会开始的更有利可图的项目保留了高水平的职员。

因为项目的成本与客户的开价呈松相关性，“根据客户预算报价”是一种普遍采用的策略。“根据客户预算报价”意思是：公司对客户期望的报价有一定的了解，继而根据客户期望的价格投标。这看似不道德而且不实事求是，然而，对于客户和系统开发者双方来说它确实有一些好处。

项目成本根据项目建议书而定。开发商和客户之间通过协商来建立详细的项目描述，该描述受到双方认可的成本的制约。买方和卖方必须对什么是可接受的系统功能取得共识。在许多项目中，不变的因素是成本而不是项目需求。为了不超出成本预算可能会改变需求。

举个例子，一个公司（OilSoft）投标为某石油公司开发一个新的油料传输系统，该系统为公司的加油服务站安排油料进度。由于没有关于该系统详细的需求文件，开发者估计 90 万美元的价格可能有竞争力，并且在石油公司的预算之内。在签订合同之后，OilSoft 公司然后协商系统的详细需求以定下基本的功能。并进而估算其他需求带来的成本。石油公司没有必要为此灰心，毕竟是将合同授予了一家可信任的公司，额外的需求可从以后的预算中支付，因此石油公司的预算也不会因为很高的软件费用而受影响。

23.2 计划驱动的开发

计划驱动的开发或者是基于计划的开发，它是一种给开发过程制定详细的计划的软件工程方法。首先是要创建项目计划。项目计划完整地记录：要完成的工作，谁将执行此工作，开发进度安排，以及项目的成果是什么。管理者使用计划支持项目决策并将其作为衡量进展的方法。计划驱动开发基于工程项目管理技术，可以看做管理大型软件开发项目的传统方法。和敏捷开发比起来，敏捷开发中会将很多影响开发的决策推迟到开发过程中做出。

反对计划驱动的开发的主要理由是：由于软件开发和使用的环境的变化，必须修改许多早

期的决策。由于避免了不必要的重复工作，延迟决策是明智的。赞同计划驱动的理由是：早期计划能够很好地考虑机构问题（机构员工等），能够在项目开始前发现潜在的问题和依赖性，而不用等到项目开发时才发现。

在作者看来，项目规划最好的方法是将计划驱动方法和敏捷开发结合起来。其中的平衡取决于项目的类型和人员的技术水平。在极端的情况下，大型信息安全和安全要求极高的系统需要大量的前期分析，在投入使用前必须万无一失。这种系统大部分应该是计划驱动的。在另一种极端情况下，用于在快速变化环境中的小型或中型信息系统应该使用敏捷方法开发。当几个公司合作开发项目时，通常使用计划驱动的方法协调各个开发地点的工作。

23.2.1 项目计划

在计划驱动的项目开发中，项目计划包括项目可用的资源、工作分解以及完成工作的进度安排。计划应该指出开发的项目和软件的风险以及用于风险管理的方法。项目计划书的具体内容随着项目和开发机构类型的不同而改变。不过，多数的计划书应该包括以下几个部分：

1. **引言** 这一部分简要论述项目的目标，并列出影响项目管理的种种约束条件，如预算、时间的限制等。

2. **项目组织** 这一部分阐述开发团队的组织方式、人员构成及其分工。

3. **风险分析** 这一部分分析项目可能存在的风险以及这些风险发生的可能性，并提出降低风险的策略。风险管理在第22章中讨论。

4. **硬件和软件资源需求** 这一部分介绍完成开发所需的硬件和支持软件。如果需要购买硬件，应注明估算的价格和交付的时间。

5. **工作分解** 这一部分把项目分解成一系列的活动，指定项目里程碑和可交付的文档，在项目中里程碑是项目的关键阶段，借此可以评估过程，可交付文档是能够交付给客户的工作产品。

6. **项目进度安排** 这一部分要描述项目中各活动之间的依赖关系。到达每个里程碑预期所需的时间以及人员在活动中的分配。本章下一节将讨论进度安排可能的表示形式。

7. **监控和报告机制** 这一部分说明要提交哪些管理报告、什么时候提交，以及使用什么样的项目监控机制。

不但要制定主项目计划，它集中分析项目风险和项目进度安排，还要制定多个辅助计划，用于支持其他过程活动，比如测试和配置管理。图23-2给出了一些可能的辅助计划的例子。

计 划	描 述
质量计划	描述在项目中所要使用的质量过程 and 标准
有效性验证计划	描述系统有效性验证所采用的方法、资源和进度安排
配置管理计划	描述所要采用的配置管理过程和结构
维护计划	预测维护需求、成本以及工作量
员工发展计划	描述如何发展项目人员的技能和经验

图 23-2 项目辅助计划

23.2.2 规划过程

项目规划是一个迭代的过程，在项目的启动阶段初始项目计划的创建就开始了。图23-3是

UML 活动图，给出了典型的项目规划过程的工作流。计划不可避免地会改变。由于在项目进行期间不断产生新的关于系统和项目团队的信息，所以必须经常性地修正原有的计划，反映出需求、进度安排以及风险的变更。业务目标发生改变，也会导致项目计划相应地改变。业务目标发生改变，会影响到所有的项目，这些项目就必须重新规划。

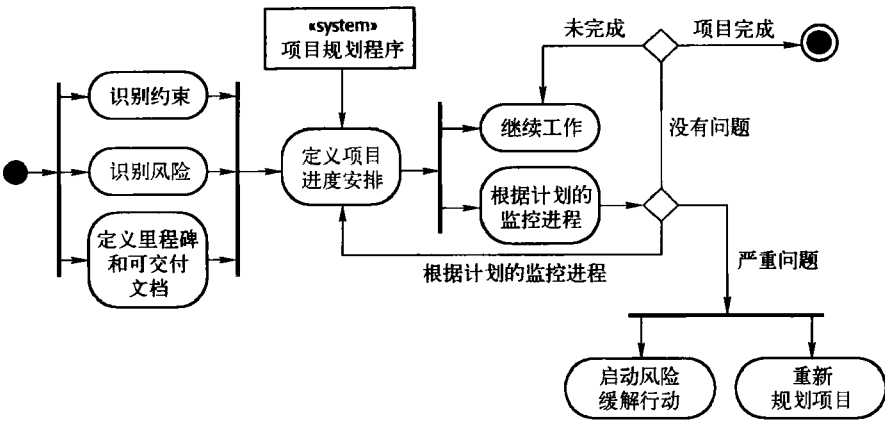


图 23-3 项目规划过程

在规划过程的开始阶段，应该从评估影响项目的各种约束条件开始。这些约束条件包括项目的交付日期、现有的人员情况、总体预算、可用工具等。另外还要定义项目的里程碑和可交付文档。里程碑是进度安排中的能够估计项目进展的那些位置点，比如移交系统进行测试。可交付文档是交付给客户的项目成果（比如系统的需求文档）。

接下去规划过程进入一个循环：拟定项目的进度安排；并启动进度安排中的各种活动或继续某些活动。在一段时间之后（通常约 2~3 个星期），检查项目的进展状况，注意项目进展与进度安排的偏差。因为最初对项目参数的估算肯定是近似的，存在小的拖延是正常的，所以计划需要不断地修改。

创建项目计划的时候尊重事实是很重要的。在一个项目中一些问题会不停地出现，这将导致项目的延期。项目开始时的设想和进度安排应该保持低调，而不能太乐观。在拟定计划时应该把各种偶然因素充分考虑进去，这样就不用每一个规划循环中都要重新协商项目的种种约束条件和里程碑等事宜了。

如果开发工作发生严重的问题，可能导致项目很大的延期，就需要采取风险缓解措施减少项目失败的风险。除了这些措施，还需要重新规划项目，可能就项目的种种约束条件和可交付的文档的有关事宜，重新与客户协商。应该重新建立新的何时完工的进度安排，并征得客户同意。

如果协商不成或者风险缓解措施没有效果，就应该安排正式的项目技术评审。评审的目标是找到能够使得项目继续进行的替代方法，并且检查项目是否与客户和软件开发者的目标保持一致。

评审（review）的结果可能是做出取消项目的决定。这可能是技术上或者管理上的失败。但是，更可能是外部变化影响项目的结果。大型软件的开发时间往往达到数年。在开发过程中，业务目标和企业优先考虑的事情不可避免地发生变化。如果这些变化表示不再需要这套软件或者原始项目需求不妥，管理人员可能决定停止软件开发，或者对项目做出重大改变以反映机构的目标变化。

23.3 项目进度安排

项目进度安排是决定如何组织项目工作，将其分割成单独的一个个任务，并且何时以何种方式完成各项任务的过程。需要估算需要用于完成各个任务的时间、需要的成本以及完成这些既定任务的人员。除此之外，还必须估算完成每项任务所需要的资源，比如服务器上需要的磁盘空间、需要在专门硬件（如仿真器等）上占用的时间以及项目人员的差旅费预算。正如本章绪论中讨论的，项目启动阶段通常会创建一个初始项目进度安排。此进度安排会在后续的开发规划过程中进一步得到修改。

基于计划的过程和敏捷过程都需要初始项目进度安排，只是敏捷项目计划不太详细。初始进度安排用于计划如何给项目分配人员，检查项目进展是否符合合同承诺。传统开发过程中，在开始阶段就创建完整的进度安排，并且随着项目进行而修改。敏捷过程中，必须有一个总的进度安排，确定何时完成项目的各个主要阶段。然后再使用迭代的方法规划各个阶段。

计划驱动项目的进度安排（见图 23-4）包括把一个项目所有的工作分解为若干个独立的任务，以及判断完成这些任务所需的时间。正常情况下项目的各项活动应该至少持续一个星期，并短于两个月。更细的划分则意味着在重新规划和更新项目计划上会花掉太多时间。对所有项目活动安排的最高的时间期限为 8 ~ 10 周左右。如果一项活动持续的时间超出这个范围，就应该在项目计划和进度安排中再次细分。



活动图

活动图是项目进度安排的一种表示方法，它给出那些能并行执行的任务和那些必须按顺序执行的任务，这些都是根据某任务对先前的任务的依赖性做出的。如果一个任务依赖于几个其他任务，那么所有被依赖的任务都必须在此任务开始之前完成。活动图中所谓的“关键路径”是最长的依赖任务序列。它决定了项目的持续时间。

[http://www. SoftwareEngineering-9. com/Web/Planning/activities. html](http://www.SoftwareEngineering-9.com/Web/Planning/activities.html)

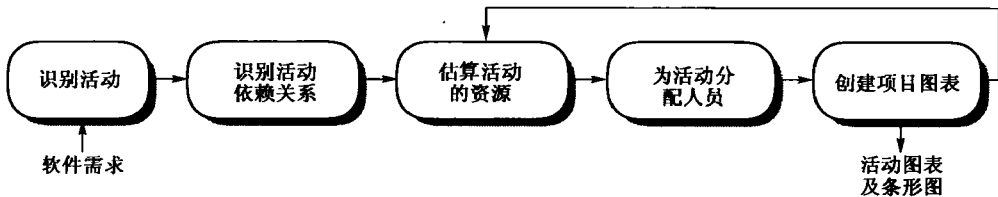


图 23-4 项目进度安排过程

有些任务是并行进行的，不同的员工研发系统不同的部件。负责进度安排的人员必须协调这些并行任务并把整个工作组织起来，从而使工作量资源得到充分利用。同时尽量避免各个任务之间不必要的依赖性。一定要避免出现因一项关键任务没有完成而使整个项目延期交付的情形。

如果一个项目在技术上非常先进，即使管理者把所有可能的意外都考虑进去，初始的估算也肯定是偏乐观的。在这一点上，软件进度安排与任何其他类型的大型高技术项目没有什么不同。新的飞机、桥梁甚至新型的汽车因为意想不到的问题常常延期交付。因此，随着有关项目进展信息的增多，必须不断地更新项目进度安排。如果进行进度安排的项目与原来某项目相似，可以沿用原来的进度安排。事实上，由于不同项目可能使用不同的设计方法和使用不同的实现语言，先前项目的经验可能不适用于规划新项目。

在估算进度时，管理者应该考虑到项目出现问题的可能性。因为做这个项目的个别人员可能生病或离职，硬件可能会崩溃，所需的基本的支持软件或硬件有可能迟迟不能交付。如果这是个新项目并且技术先进，其中某些部分可能比原来预期的要困难得多，花费的时间也多。

一个好的经验法则是进行估算时先假定什么问题也没有，然后再把预计出现的问题加到估算中去。其他的偶然因素可能带来意想不到的问题，在估算时也可以考虑进去。这些偶然因素是由项目的类型、过程参数（截止期限、标准等）以及该项目的软件工程人员的素质和经验决定的。意外情况可能使项目需要的成本和时间增加 30% ~ 50%。

进度安排表示方法

项目进度安排可简单地用一个表来表示。列出任务、工作量、计划工期、任务依赖性（见图 23-5）。但是这种表示方式很难发现不同活动之间的关系和依赖性。所以，人们研究出了另外一些更容易阅读和理解的图形表示方法。通常使用的有以下两种表示方法：

- 1. 条形图，基于日历时间，条形图可以表示每项活动的负责人是谁，预计所用的时间，以及该项活动预计的开始和结束时间。条形图有时也叫甘特图，是以发明人 Henry Grantt 命名的。
- 2. 活动网络图，这是一个网络图，表示构成项目的不同活动之间的依赖关系。

任 务	工作量（人日）	工期（天）	依 赖 关 系
T1	15	10	
T2	8	15	
T3	20	15	T1（M1）
T4	5	10	
T5	5	10	T2，T4（M3）
T6	10	5	T1，T2（M4）
T7	25	20	T1（M1）
T8	75	25	T4（M2）
T9	10	15	T3，T6（M5）
T10	20	15	T7，T8（M6）
T11	10	10	T9（M7）
T12	20	10	T10，T11（M8）

图 23-5 任务、持续时间以及依赖关系

通常情况下，使用项目规划工具管理项目进度安排信息。使用这些工具，只需要将项目信息输入在表格中，就会生成项目信息数据库。条形图和活动网络图从项目信息数据库中自动生成。

项目活动是基本的规划单元，每个活动有：

- 1. 以天或者以月计算的工期。
- 2. 工作量估计，反映完成工作所需的人日或人月数。
- 3. 活动完成的期限。
- 4. 定义好的终点，表示完成一项活动的明确结果。可以是一份文档，举行评审会，或者是所有测试的成功完成等。

在进行项目规划时，应该建立项目里程碑，一个里程碑就是项目中的每个阶段，通过它能够对项目过程进行评估。每个里程碑应该用一个简短的报告总结取得的进展和完成的工作。里程

碑可能是关于单个的任务或者是一组相关联的活动。比如，在图 23-5 中，里程碑 M1 和任务 T1 有关，而里程碑 M3 和两个任务 T2、T4 有关。

项目可交付的文档是一种特殊的里程碑。这是交付给客户的项目成果。通常在项目的描述、设计等主要的项目阶段结束时交付。通常情况，可交付的文档是在项目合同中明确说明的，在客户看来项目的进展表现为这些可交付的文档。

为了说明条形图的使用，我们创建一个假想的一组任务，如下面的图 23-5 所示。图中给出了各项活动、估计的工作量投入、持续时间以及和任务之间的相互依赖关系。从图 23-5 可以看出任务 T3 依赖于任务 T1，也就是说 T1 必须要在 T3 开始前完成。举例来说，T1 可能是组件设计的准备活动，而 T3 就是该设计的实现，要想开始实现该设计，应该首先完成这项设计。可以看出一些活动估计的工期比必需的工作量长；而有些则相反。如果工作量比工期少，这就表示分配到这个任务的人员没有全职工作。如果工作量超过工期，就表示几个成员同时进行这个任务。

图 23-6 是使用图 23-5 中的信息以图形的形式表示项目的进度安排。它是一种条形图，表示了项目的日程安排和各项任务的开始和完成日期。从左往右阅读，条形图清晰地表示任务何时开始以及何时结束。条形图上也标明了里程碑（M1、M2 等）。可以看出独立的任务是并行执行的（比如任务 T1、T2 和 T4 都在项目的起始处开始执行）。

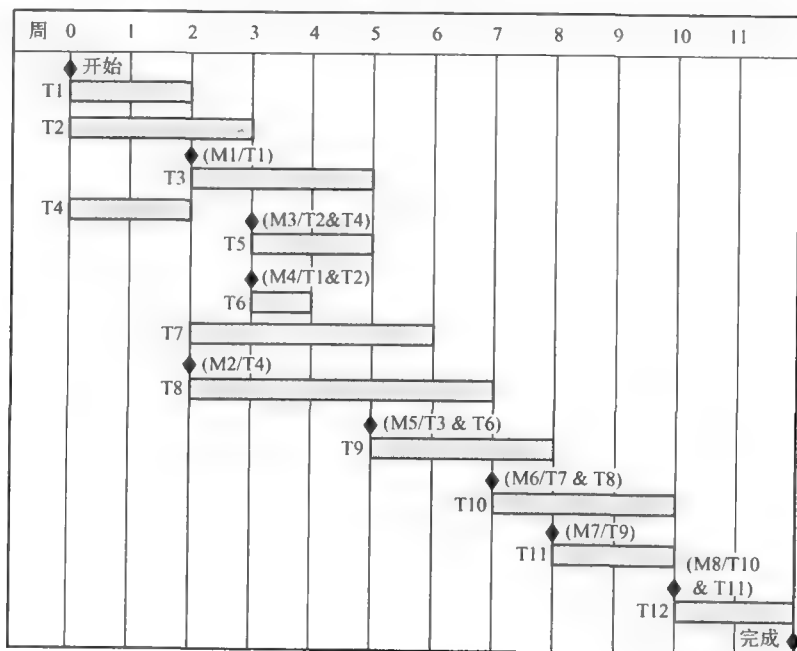


图 23-6 活动的条形图

除了考虑进度安排外，项目管理者还要考虑资源的分配，尤其是参加项目活动的人员的分配，以及给他们分配项目任务。这种分配也要作为项目管理工具的输入，可以生成条形图，从而表示出在哪些时间段上雇用哪些职员（见图 23-7）。项目职员可能同时承担多个任务，或者有时他们并不在项目中工作。在这期间他们可以休假、做别的项目、参加培训或进行其他的活动。图中的兼职任务使用对角线在任务条上标识。

大型的开发机构通常会根据项目需要聘请许多专家，这有可能影响项目的进度安排。在图 23-7 中，我们可以看到，Mary 是一位专家，他只参加项目中的一个任务。这有可能引起进度安排问题。如果一个有专家参与的项目延迟了，就会给其他也需要专家的项目带来连锁反应。因为

专家无法到位，这些项目很可能也要延迟完成。

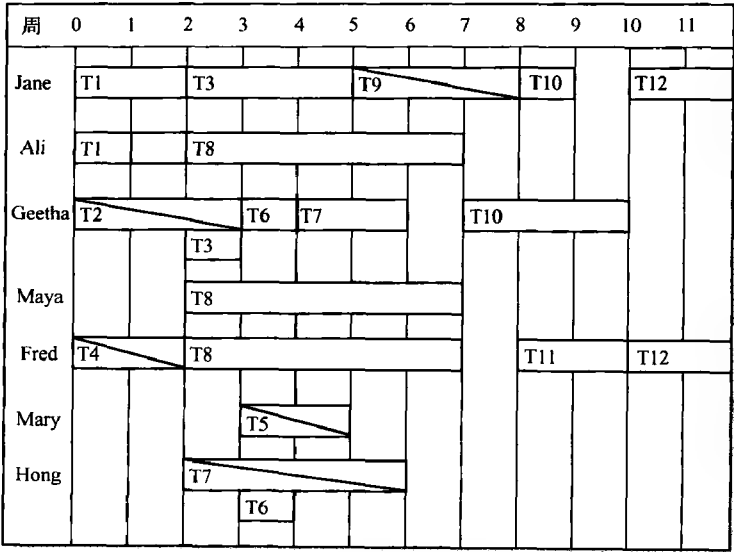


图 23-7 人员分配示意图

如果任务延期了，将明显地影响以后的依赖于它的其他任务。只有延期的项目完成后，其他项目才能开始。任务延期会引起严重的人员分配问题，特别是在人员同时参与几个项目的情况下。如果任务 T 延期，可能将分配给任务 T 的人员分配其他工作（任务 W）。完成任务 W 可能比任务 T 的时间长，但是人员一旦分配出去，就不可能轻易地将他们重新分配回原来的任务（T）。由于等待任务 W 的完成，这将进一步导致任务 T 的延期。

23.4 敏捷规划

软件开发中的敏捷方法是一种迭代的方法，软件是以增量式方式进行开发和交付给客户。和计划驱动方法不一样的是，这些增量中的功能不是提前计划好的，而是在开发过程中决定的。决定增量包中包含何种功能取决于进展情况以及客户的需求的优先级。选择这个方法的理由是：客户需求的优先级和需求经常改变，所以制定能适应这些变化的灵活的计划是合理的。Cohn 的书中 { Cohn,2005##1735 } 全面讨论了敏捷方法中的项目规划问题。

最常用的敏捷方法，比如 Scrum (Schwaber, 2004) 和极限编程 (Beck, 2000)，它的规划是两个阶段法，这两个阶段分别对应着计划驱动开发的启动阶段和开发规划：

- 1. 版本规划，包括对未来几月的展望以及决定系统的发布版本中应该包含的功能。
- 2. 迭代规划，包括短期的展望，主要是规划系统的下一个增量。一般这将花费整个团队 2 ~ 4 个星期的工作量。

第 3 章讨论了 Scrum 规划方法，接下来重点介绍极限编程 (XP) 规划。这种方法叫做“规划游戏”，参与人员包括整个开发团队以及客户代表。图 23-8 表示了规划游戏的各个阶段。

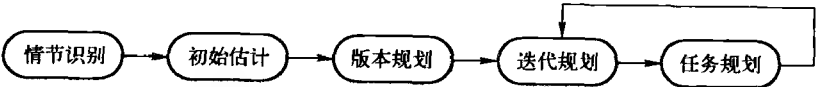


图 23-8 XP 中的规划

XP 中的系统描述是基于用户情节的，用户情节反映了应该包含在系统中的特征。在项目启动的时候，开发团队和客户试着定义一系列的情景，使其能覆盖最终系统中包括的所有功能。不可避免的是，肯定会遗漏一些功能，但是在这个阶段这并不重要。

下一阶段是评估阶段。项目组阅读并且讨论这些情景，然后按照实现这些情景所需要的时间将情景排序。这可能包括将大的情景拆分成较小的情景。相对评估经常比绝对估计更容易。人们经常发现评估做某些事情所需要的工作量和时间是很困难的。然而，当几件需要完成的事情摆在面前的时候，人们能够判断哪件事会花费最多的时间和工作量。排序完成以后，项目组给这些情景分配理论工作量点。一个复杂的情景需要 8 个点，简单的需要 2 个点。将序列中所有情景都标上相应的理论工作量点。

进行了情景评估之后，使用一个“速度”的概念，将相对工作量转变成对所要求的总体工作量的第一次估计。在 XP 中，速度是项目组每天所能完成的工作量点数量。这可以通过之前的经验估计或者实现一两个情景了解需要多少时间。速度估计虽然是近似的，但是可以在开发过程中进一步精练。一旦有了速度估计，就能够以人日数的形式计算实现整个系统所需的工作量。

发布规划包括选择和完善上述情景，这些情景反映出了在系统的发布版本中应实现的功能以及实现这些情景的顺序。客户应该参与这个过程。接下来选择一个发布日期，检查情景以判断工作量估计是否满足发布日期。假如不满足的话，增加或者删除清单上的一些情景。

迭代规划是迭代开发过程的第一步。选择迭代过程要实现的情景，情景的个数反映了交付一次迭代的时间（通常为 2~3 周）和项目组的速度。迭代交付日期到达之时，即使并不是所有情景都已实现，这次迭代也宣告完成。项目组考虑已实现的情景，增加其工作量点。重新计算速度，将其用于下一个系统版本的规划。

每次迭代的开始，会有一个更详细的规划阶段，开发人员将情景拆分成各个开发任务。一项开发任务大概使用 4~16 小时。列出所有这次迭代中的必须完成的任务。每个开发者申请他们要完成的任务。每个开发者知道自己的开发速度，不能申请比他们能在规定时间内能完成的更多的任务量。

这个任务分配方法有如下两个主要的好处：

1. 整个项目组对迭代过程中要完成的任务有一个整体认识。因此他们能够理解项目组其他成员的工作内容以及确定任务依赖关系后应和谁交流。
2. 每个开发者选择要完成的任务；并不是简单地由项目管理者分配任务。这样开发者对自己选择的任务有一种拥有感，就可能激发他们更好地完成任务。

迭代过程进行到一半的时候，进行进展评审。这时，应该已完成一半的情景工作量点。所以，如果一次迭代包含 24 个情景点和 36 个任务。应该已完成 12 个情景点和 18 个任务。如果没有完成的话，必须征询客户的意见删除迭代中的一些情景。

这种规划方法有一个好处：软件一直处于发布和规划之中，没有进度安排。如果不能按时完成工作，XP 的观点是减少工作量，而不是延长进度安排。然而在某些情况下，所发布的增量可能不能满足当前用户使用的要求。减量会给客户带来额外负担，因为他们不得不使用不完整的系统或者因在两个系统版本之间的切换改变了他们的工作方式。

敏捷方法中的主要困难在于它依赖于客户参与。实际上，由于客户代表必须先忙于其他工作，这很难安排。客户可能更加熟悉传统项目规划，也许很难参与敏捷规划项目。

对于那些能聚在一起讨论要实现的情景的小型、稳定的开发团队而言，敏捷方法能够很好地运作。但是对于那些庞大并且分布在不同地点的项目组或者组员频繁变动的项目组而言，实际上不可能每个人参与到敏捷项目管理中最为核心的集体规划中。所以，通常使用传统软件

管理方法对大型项目进行规划。

23.5 估算技术

项目进度估算非常困难。初始的估算可能需要根据高层的用户需求定义做出。软件可能需要运行于某些特殊类型的计算机上,或者需要运用到新的开发技术。对参与到项目中来的人员的技术水平可能还一无所知。如此多的不确定因素意味着,在项目早期阶段对系统开发成本进行精确估算是相当困难的。

评估用于成本和工作量估算的不同方法的精确性有它固有的困难。项目估算通常是先入为主的。项目估算是用来确定项目预算,然后通过调整产品以保证预算不被突破。为了让项目开支控制在预算之内,可能会牺牲掉一些待开发的软件特性。

作者的经验是:任何一个受控的项目成本估计实验,其间都受到成本的左右。受控实验不会向项目管理人员揭示出成本估算。实际的成本将与估计的项目成本进行比较。尽管如此,机构还是需要软件所需工作量和成本进行估算的。有以下两种类型的估算技术:

1. **基于经验的技术** 使用管理者之前项目和应用领域的经验估算要求的未来工作量,即管理者主观给出所需要的工作量的一个估计。

2. **算法成本建模** 在此方法中,使用一个公式方法计算项目的工作量,它基于对产品属性(比如规模)和过程特点(比如参与员工的经验)的估计。

无论以上哪种技术,都需要使用眼力直接估算工作量或者估算项目和产品特点。在项目的启动阶段,估计的偏差比较大。基于从大量项目中收集的数据,Boehm等(1995)发现启动阶段的估算差异巨大。假如开始的工作量估计是 x 个月,那么系统交付时测量的实际工作量范围可能是 $0.25x \sim 4x$ 之间。在开发规划中,随着项目的进行估算会越来越准确(见图23-9)。

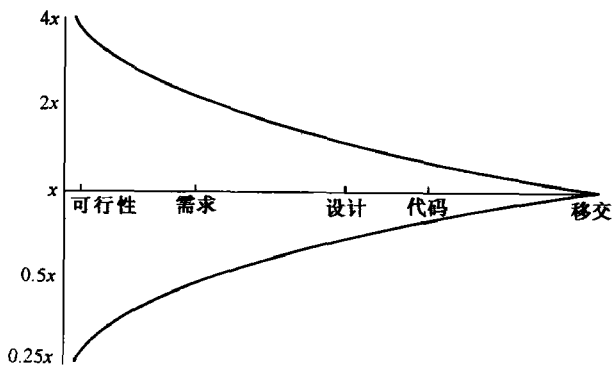


图 23-9 估算的不确定性

基于经验的技术依赖管理者之前项目的经验以及这些项目中有关软件开发活动上的实际工作量投入。通常,项目管理者需要定义项目中要生成的可交付

物以及不同的软件组件或者要开发的系统。将这些内容记录在电子表格中,单独估算各个子项,再计算需要的总工作量。通常召集一组人参与估算工作量,并要求每个组员解释各自的估算结果,这样做很有帮助。很多情况下这将暴露别人没考虑到的因素,管理者重复这个过程谋求一个达成共识的估算结果。

基于经验的技术的困难在于一项新软件项目可能和之前的项目没有太多的共同点。软件开发发展非常迅速,项目经常使用一些不熟悉的技术,比如页面服务,基于COTS的开发,或者AJAX。如果管理者没有使用过这些技术,之前的经验可能对估算需要的工作量没有帮助,从而使准确的成本和进度估算更加困难。

23.5.1 算法成本建模

算法成本建模基于项目的规模、开发的项目类型、其他团队、过程和产品因素,用一个数学公式预测项目的成本。算法成本模型可以通过对已完成项目的成本及其特性的分析建立起来,并找到最接近的公式适应实际情况。

算法成本模型主要用于估算软件的开发成本，但 Boehm 以及他的合作者（2000）讨论了这些模型的另外一些用途，比如软件公司投资者估算的准备，帮助评估风险的可选策略，以及关于经验性的复用、再开发或外购的决策。

软件项目中的工作量估算的算法模型可基于一个简单的公式：

$$\text{Effort} = A \times \text{Size}^B \times M$$

A 是一个常数因子，依赖于机构的实践经验和所开发的软件类型。Size 可以是软件的代码行数或是用功能点或应用点表示的功能的估算。指数 B 的值通常在 1 ~ 1.5 之间，M 是一个乘数因子，反映了过程、产品、开发属性，例如软件的可依赖性需求，以及开发团队的经验等综合因素。

交付的系统中的源代码的行数（SLOC）是基本的规模度量标准，可以用于许多的算法成本建模。规模估计包括类比其他项目的估计，把函数或者应用点数转化成代码多少的估计，将系统组件按规模排序并使用已知参照组件估计组件规模，或者只是简单地从一个提问获得对此的工程判断。

绝大多数算法估算模型都有指数成分（上式中的 B），这与规模和系统的复杂度有关。这反映了一个事实，那就是，成本一般都不是与项目规模呈线性关系的。随着软件规模和复杂度的增大，扩大的团队的通信费用在增加，需要的配置管理更复杂，系统的集成难度也在加大，所有这都要付出额外的费用。系统越复杂，这些因素影响成本越多。因此，通常 B 的值随着系统的规模和复杂度增加。

所有的算法模型都存在同样一些难题：

1. 在项目早期阶段只有描述存在的情况下，估算 Size 值通常是个难题。功能点和应用点估算（后面将谈到）比估算代码长度要容易，但是不够精确。
2. 因子 B 和 M 的估算往往带有主观色彩。对其估算因人而异，这与一个人的经验和背景有关。

准确的代码规模估算在项目早期是很难的，因为最终程序规模依赖于设计决策，而需要估算的时候决策还未形成。举例来说，一个需要复杂数据管理的应用，可以使用自己实现的数据管理工具，也可以使用商业数据库。在最初的成本估计时，不可能知道是否存在表现足够好、能够满足运行需求的商业数据系统。所以不知道要在系统中加入多少数据管理代码。

系统开发所使用的程序语言也会产生重要影响。使用像 Java 这样的语言较之使用 C 语言，可能意味着需要较多的程序代码。不过，这些额外的代码会带来更多编译时检查，所以检验成本就会降低。如何考虑这些因素？此外，在软件开发过程中复用的程度也一定要估算，由此修正对 Size 值的估算。

算法成本模型是一种估算开发一套系统所需工作量的系统方法。然而，这些模型很复杂而且很难使用。模型中有很多的属性，在估算它们的值的时候有相当大的不确定性。这种复杂性阻碍了许多潜在的用户，因此只限于一小部分公司实际应用算法成本模型。

使用算法模型的另一个障碍是需要校准。模型用户应该根据他们自己的历史项目数据校准他们的模型和参数值，因为这反映了部门实际和经验。但是几乎没有机构从过去的项目中收集了足够的数据，这种数据会以一种表格的形式存在以支持模型校准。因此，算法模型的实际使用必须以模型参数的发布值开始。实际上建模者不可能知道这些是如何紧密地关联到他们自己的机构的。

如果算法模型用于项目成本计算，需要用所估算的项目类型数据进行校正，对于模型的输出也要慎重解释。估算者应该做一系列估算（最坏估算、期望估算和最好估算）而不是单一估算，并用成本计算公式都计算一遍。初始估算中的错误有可能是相当大的。只有在对产品已经非常了解，所使用的模型经过机构的长期使用，其参数已经校正得较为准确，而且语言和硬件选择

都预先确定了的情况下，成本估算才最有可能取得精确的结果。

23.5.2 COCOMO II 模型

人们已经提出几个相似模型用来帮助估算工作量、进度和成本。本节专门讨论 COCOMO II 模型。COCOMO II 模型完全是一个经验模型。获得该模型的途径是这样的：首先从大量软件项目中收集数据，然后通过对这些数据的分析找出与观察资料最相符合的公式。这些公式将系统的规模、产品、项目和团队因素等与开发系统的工作量联系起来。COCOMO II 是一个良好文档化的、非专利估算模型。

COCOMO 模型很大部分基于原代码开发 (Boehm, 1981; Boehm 和 Royce, 1989), COCOMO II 模型是从早期的 COCOMO 模型开发而来。COCOMO II 模型在软件开发中考虑到更多的现代方法，比如使用动态语言的快速开发、组件组合开发，以及使用数据库编程。COCOMO II 模型支持开发的螺旋式模型，见第 2 章，且内嵌了一些进行渐进深入的估算的子模型。

作为 COCOMO II 模型一部分的子模型（见图 23-10）有：

1. **应用组合模型** 它对所需的工作量建模，所开发的系统是由可复用组件、脚本或数据库编程创建而得。软件规模估算基于应用点，还有个简单的规模/生产率公式用于估算所需的工作量。程序中的应用点是一种加权估计，对以下几项进行加权计算：单独的显示屏幕的数量，生成的报告的数量，命令式编程语言（比如 Java）中的模块数量，脚本语言或者数据库编程代码的行数。

2. **早期设计模型** 这个模型用于在得到需求之后的早期系统设计阶段。估计是基于在引言中谈到的标准估计公式，一组包含 7 个因子的简化了的参数。使用功能点进行估算，然后转化为源代码的行数。功能点是一种独立于语言的量化程序功能的方式。通过衡量或者估算外部输入和输出的数量、用户迭代的次数、外部界面的数量，以及系统使用的文件或数据库的数量，计算出程序中的所有功能点数目。

3. **复用模型** 这个模型用于计算集成可复用组件和/或由设计或自动生成的程序代码所需的工作量。它一般与后体系结构模型结合使用。

4. **后体系结构模型** 当设计出系统体系结构后，就可以对软件规模做更精确的估算。这个模型也使用上面谈到的成本估算的标准公式，但它包含了更广泛的有 17 个反映个人能力和产品与项目特征参数集合。

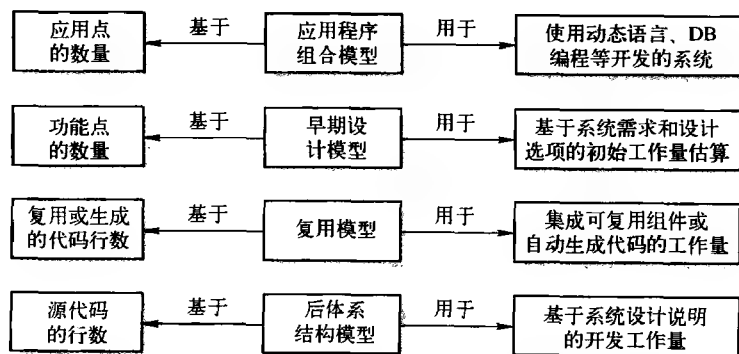


图 23-10 COCOMO 估计模型

当然，在大型系统中，不同部分可能使用不同的技术进行开发，不要求以同等的精确度估算系统的所有部分。在这种情况下可以为每个部分采用适当的子模型，然后合并这些结果得到一个合成的估算值。



软件生产率

软件生产率是对软件工程师在每周或每个月完成的开发工作的一个平均数量的估计。因而它表达为每月代码行数，每月功能点数等。

然而，在具有有形的结果（例如，办事员每天处理 N 个票据）的地方，生产率是容易计算出来的，而软件生产率相对而言就十分难于定义。不同的人对于相同的功能会用不同的方式实现，结果他们使用的代码行数就会不同。代码的质量也是重要的，但是在某种程度上，是主观性的。因而比较不同软件工程师间的生产率是很不可靠的，因此对项目规划没有太大的用处。

<http://www.SoftwareEngineering-9.com/Web/Planning/productivity.html>

应用 - 组合模型

将应用 - 组合模型引入 COCOMO III 以支持对项目所需要的工作量进行估算。此模型适合于原型构造型项目和通过已有的组件组合进行软件开发的项目。它的计算是基于加权的应用点（有时也称为对象点）除以一个标准应用点生产率。然后按照开发每个应用点（Boehm 等，2000）的难度对估算值进行调整。程序员的生产率取决于开发者的经验、能力以及所使用的软件工具（ICASE）的水平。图 23-11 是 COCOMO 模型开发者给出的不同应用点生产率水平（Boehm 等，1995）。

开发者的经验和能力	非常低	低	一般	高	非常高
CASE 工具的成熟度和能力	非常低	低	一般	高	非常高
PROD (NOP/月)	4	7	13	25	50

图 23-11 应用点生产率

应用合成通常包括大量的软件复用，很肯定的是系统里应用点中的一部分可能由可复用组件来实现，这就需要考虑到预期复用部分所占的比例来调整基于应用点总数的估算。因此，对于原型系统开发项目的工作量计算的最后公式是：

$$PM = (NAP \times (1 - \% reuse/100)) / PROD$$

其中，PM 是以人月为单位的工作量，NAP 是交付系统的应用程序点的总数，% reuse 是在开发中利用的代码量的估计，PROD 是如图 23-11 所示的应用点生产率。这个模型产生的估计只是近似情况，它并未考虑复用中的额外工作量。

早期设计模型

在项目的初始阶段使用这个模型，此时还未对系统体系结构做出详细的设计。早期设计估计对于选择探索是最为有用的，此时我们需要比较实现用户需求的各种方式。早期设计模型假定用户需求已经确定以及系统设计过程的初始阶段已经开始。这一阶段的任务应当是简单快速地做一个大略的成本估计。因此需要做出各种简单化的假设，如集成可复用代码的工作量为零。

在这一阶段做出的估算是基于算法模型的标准公式得出的，即

$$Effort = A \times Size^B \times M$$

Boehm 基于他自己的大型数据集提出对于这一阶段的估算，系数 A 应该为 2.94，系统的规模用 KSLOC 表示，即源程序代码以千行为单位的数量。这是通过估算软件中功能点数，然后使用标准表格将功能点转换成 KSLOC 来计算的，这个标准表格针对不同语言，给出功能点和软件规模的对应值。

指数 B 反映了随着项目规模的扩大所需工作量的增长。它可以在 1.1 ~ 1.24 之间变动，与项目的创新程度、开发的灵活性、采用的风险处理过程、开发团队凝聚力，以及机构的过程成熟度（见第 26 章）水平等密切相关。在关于 COCOMO II 的后体系结构模型中会描述用这些参数计算该指数的方法。

由此得到下列工作量计算公式：

$$PM = 2.94 \times \text{Size}^{(1.1 \sim 1.24)} \times M$$

这里

$$M = \text{PERS} \times \text{RCPX} \times \text{RUSE} \times \text{PDIF} \times \text{PREX} \times \text{FCIL} \times \text{SCED}$$

乘数因子 M 基于 7 个项目和过程属性，这可以增加或减少估计。用在早期设计模型中的这些属性分别是：产品可靠性和复杂性（RCPX），要求的复用数（RUSE），平台困难程度（PDIF），个人能力（PERS），个人经验（PREX），进度（SCED）以及支持设施（FCIL）。本书的网页上有对这些属性的解释。对这些因子的取值可以分成六档，“非常低”的值该因子赋值 1，对具有“非常高”的值该因子赋值为 6。

复用模型

如在第 16 章所述，软件复用目前很普遍，大多数大型系统都有很大一部分代码是从以前开发的系统中复用而来。复用模型用于估计集成可复用代码或已生成代码所需的工作量。

COCOMO II 将复用的代码分为两类。黑盒代码是那种不需去理解或做出改动的代码，对黑盒代码不需要进行开发工作。白盒代码需要将新代码或其他复用组件的代码改编整合在一起，它的复用需要一些开发工作，因为需要理解并修改以使它在系统中正常工作。

许多系统包括根据系统模型自动生成的代码，见第 5 章。模型（通常使用 UML 描述）经过分析，模型中定义的对象生成为代码。COCOMO II 复用模型包括一个用于估算集成这些生成的代码要求的工作量的公式：

$$PM_{\text{Auto}} = (\text{ASLOC} \times (\text{AT}/100)) / \text{ATPROD} // \text{生成代码的估计}$$

式中：ASLOC 是复用代码的总行数，包含自动生成的代码。AT 是自动生成的复用代码所占的百分比。ATPROD 是工程师在集成这些代码时的生产率。

Boehm 等人（2000）估计 ATPROD 为每月 2400 条源码语句。因此，如果在系统中有 20 000 行复用代码且其中 30% 是自动生成的，则整合这些生成代码的工作量是：

$$(20\,000 \times 30/100) / 2400 = 2.5 \text{ 人月} // \text{生成代码的例子}$$

可使用另一个计算方法估算整合这些其他系统的复用代码需要的工作量。复用模型不直接从复用组件的数量估计计算工作量，而是根据复用的代码行数计算等价于新代码行数的量。模型为计算等价的新代码（ESLOC）行数提供了一个基础。这基于需要改变的复用代码的行数以及一个能反映复用这些组件需要做的工作量的乘数。用于计算 ESLOC 的公式考虑了软件理解、对复用代码的修改，以及修改系统以集成这些代码所需的工作量。



COCOMO II 成本形成因素

COCOMO II 成本形成因素是反映产品、团队、过程以及机构因素的某些属性，这些属性影响软件系统开发中所需的工作量。例如，如果需要高级别的可靠性，那就需要投入额外的工作量；如果有快速交付的需要，那同样会有额外的工作量；如果团队成员变化了，也会需要额外的工作量。

在 COCOMO II 模型中有 17 个这样的属性，此模型的提出者给这些属性赋予了特定的值。

<http://www.SoftwareEngineering-9.com/Web/Planning/costdrivers.html>

下面的公式用于计算等价的源代码行数：

$$ESLOC = ASLOC \times AAM$$

式中：ESLOC 是新源代码的相当行数。ASLOC 必须修改的组件的代码行数。AAM 是改写调整因子，下面讨论。

复用是有代价的，即使在不改善原代码的情况下也会产生一些开销。但是随着复用代码量的增加复用的开销在降低。固定的理解和评估的成本分布到了更多的代码行上。改写调整因子（AAM）调整估计以反映复用代码需要的额外的工作量。简单来说，AAM 是 3 个部分之和：

1. 改写部分（称为 AAF），表示修改这些复用代码的成本。改写部分包括设计、编码和集成等修改几个子部分。

2. 理解部分（称为 SU），表示理解要复用的代码以及工程师熟悉这些代码所需的代价。SU 的范围为 50 ~ 10，50 表示复杂的非结构化代码，10 表示书写良好的面向对象的代码。

3. 评估因子（称为 AA），表示复用的决策成本。AA 包括决定代码是否能复用的必要的一些分析代价。根据所需的分析工作量的不同 AA 的值从 0 变化到 8。

如果能自动完成一些代码改写，将减少需要的工作量。因此通过估计自动调整的代码（AT）的百分比调整估算，并使用 AT 调整 ASLOC。因此最终公式为：

$$ESLOC = ASLOC \times (1 - AT/100) \times AAM$$

一旦计算出 ESLOC，就可以应用标准估算公式计算总工作量，规模参数等于 ESLOC。接下来将这加到已计算完的集成自动产生代码的工作量，就可以计算出需要的总工作量。

后体系结构层

后体系结构模型是 COCOMO II 模型中最详细的一个。当系统的初始结构设计完成，已知子系统结构后就会用到它。然后评估系统的每个部分。

在后体系结构层产生的估算所依据的基本公式与早期设计估算中的基本公式相同：

$$PM = A \times Size^B \times M$$

到这个阶段为止，因为知道如何将系统分解为各个对象或者模块，软件规模的估算在这个阶段中要准确得多。估算代码规模需要下面 3 个参数：

1. 要开发的新代码行数的估计（SLOC）；
2. 基于用复用模型计算得到的等价代码行数（ESLOC）的复用成本的估计；
3. 由于需求变更可能要修改的代码行数的估计。

这 3 个估计值加起来得到总的代码规模 KSLOC，用在工作量计算公式中。最后一个估计值——要修改的代码行数——反映了软件需求经常会变化。应该考虑到这将引起重复工作以及开发额外的代码。当然这个数值常常比要开发的新代码的估计有更多的不确定性。

在工作量计算公式中的指数项与项目的复杂程度有关。当项目较复杂时，增加系统规模的工作量耗费明显增大。不过，机构的实践经验和规程能控制这种因规模增加的费用，这种费用也是增加复杂度的结果。通过分析如图 23-12 所示的 5 个因子来计算指数 B 的值。这些因子都有从 0 ~ 5 六个等级，0 表示“非常高”5 表示“非常低”。计算 B 时，将这些估算值相加再除以 100，然后再加上 1.01 就是该指数项的取值了。

比如，我们假定一个机构正承担一个项目，机构对于该项目所在领域没有经验。项目客户没有定义需要采用的过程，在项目进度中也没有安排重大风险分析，而且还需要组织一个新的开发团队来完成这个系统。该机构最近刚实行过程改善计划，并且依据 SEI 能力评估模型，如第 26 章中讨论的。已经被评定为 2 级机构。在进行指数计算时用于评级的可能取值如下：

级别因素	解 释
有先例可援引	反映机构在此类型项目上先前所获得的经验。非常低意味着没有先前经验；非常高意味着机构非常熟悉此应用领域
开发的灵活性	反映开发过程的灵活性级别。非常低意味着使用一个预先指定的过程；非常高意味着客户只设定了总目标
体系结构/风险解决方案	反映了所执行的风险分析的程度。非常低意味着几乎没有分析；非常高意味着完全彻底的风险分析
团队凝聚力	反映了开发团队成员彼此了解和一起工作有多好。非常低意味着交互存在很大困难；非常高意味着一个团结高效的团队，没有沟通障碍
过程成熟度	反映了机构的过程成熟度。该值的计算依赖于 CMM 成熟度调查问卷，但是可以通过从 5 中减去 CMM 过程成熟度级别来获得一个估计

图 23-12 在后体系结构模型中指数计算所用的级别因素

1. 有先例可援引 取值为“低”（4）。这是机构的一个新项目。
2. 开发的灵活性 取值为“非常高”（1）。开发过程没有客户介入，所以几乎不存在外部强加的改变。
3. 体系结构/风险解决方案 取值为“非常低”（5）。没有风险分析。
4. 团队凝聚力 取值为“一般”（3）。是个新团队，没有相关信息。
5. 过程成熟度 取值为“一般”（3）。有一些过程控制。

这些值的总和为 16。然后除以 100 计算指数，结果 0.16 加上 1.01，所以得到调整后的指数是 1.17。

精练总工作量估计，使用 17 个因素的大集合，包括产品、过程、机构属性（成本形成因素），而不是早期设计模型中使用的 7 个属性。由于掌握了更多的软件本身的信息、非功能的需求、开发团队以及开发过程，就可以估算这些属性的值了。

图 23-13 给出一个例子，说明这些成本形成因素属性是如何影响工作量估算的。在这里对指数所取的值为 1.17，与上一个例子相同。假设指标 RELY、CPLX、STOR、TOOL 和 SCED 是项目中的关键性成本形成因素。所有其他因素取标称值 1，不会影响工作量计算。

指数值	1.17
系统规模（包括了复用和需求易变性因素）	128 000 DSI
不计成本形成因素的初始 COCOMO 估计	730 人月
可靠性	非常高，乘数 = 1.39
复杂度	非常高，乘数 = 1.3
内存限制	高，乘数 = 1.21
工具使用	低，乘数 = 1.12
进度	加速的，乘数 = 1.29
调整的 COCOMO 估计	2306 人月
可靠性	非常低，乘数 = 0.75
复杂度	非常低，乘数 = 0.75
内存限制	无，乘数 = 1
工具使用	非常高，乘数 = 0.72
进度	正常，乘数 = 1
调整的 COCOMO 估计	295 人月

图 23-13 成本形成因素对工作量估算的影响

在这个例子中，对关键性的成本形成因素分别赋予了最大和最小值来说明它们对工作量估算带来的影响。所取的值来自 COCOMO II 的参考手册（Boehm, 2000）。从图中可以看出，对成本形成因素取较高值时的工作量估算是初始估算的 3 倍多，而对这些成本形成因素取低值时的工作量估算会降低到初始估算的大约 1/3，这样就突出表示了不同类型项目之间的巨大差异，以及在将一个领域的经验移植到另一个领域时的巨大困难。

23.5.3 项目的工期和人员配备

项目管理者除了要对软件系统开发所需成本及其工作量做出估算以外，还必须估算软件的开发周期以及人员要在什么时间到位。机构不断地想缩短开发进度，希望自己的产品能赶在竞争对手之前投放市场。

COCOMO 模型包括一个对项目所需日历时间的计算公式：

$$TDEV = 3 \times (PM)^{(0.33 + 0.2 \times (B - 1.01))}$$

式中：TDEV 是项目的理论进度（月），忽略关于项目进度的任何乘数。PM 是使用 COCOMO 模型计算的工作量。B 是在 23.5.2 节讨论的复杂度相关的指数。如果 $B = 1.17$ ， $PM = 60$ ，那么：

$$TDEV = 3 \times (60)^{0.36} = 13 \text{ 月}$$

然而，使用 COCOMO 模型预测的理论工程进度和项目计划要求的进度不一定是一回事。可能需要比理论进度显示的日期提前或者延迟（比较少见）交付软件。如果时间进度被压缩，这会增加项目需要的工作量。工作量估算中的 SCED 乘子能解决此问题。

如上所述，假定项目估计 TDEV 是 13 个月，但是实际进度要求是 11 个月。这相当于将进度压缩大概 25%。使用由 Boehm 小组得到的 SCED 乘子的值，这种进度压缩的工作量乘子是 1.43。因此，如果加速后的进度能够实现，实际交付软件需要的工作量多于理论进度要求的工作量的 50%。

项目所需的工作人数、需要的工作量以及项目的交付日程之间关系非常复杂。如果 4 人能在 13 个月内完成一个项目（52 人月的工作量），也许你会想到增加一个人，就能在 11 个月内完成工作（55 人月的工作量）。但是 COCOMO 模型显示，实际上需要 6 个人才能在 11 个月内完成工作（66 人月工作量）。

这种情况的原因是增加人员实际上降低了原有组员的生产率，所以实际增加的工作量要少于一个人。当项目组规模增加的时候，组员花费更多的时间交流以及定义由别人开发的系统各个部分的接口。人员数量翻一番（打个比方），项目工期并不是简单地缩减一半。如果开发团队规模很大，有些时候情况是增加项目人员增加了开发日程而不是减少了日程。Myers（1989）讨论了加快进度问题，并认为，如果软件开发得不到充足的时间保证是很容易出大问题的。

这个 COCOMO 模型的一个有趣现象是完成项目所需的时间是项目所需的总工作量的函数，而它与项目中的软件工程师人数无关。这也验证了一点，在落后于进度的项目中增加更多的人员未必对重新赶上进度有帮助。

单纯用项目所需工作量除以开发所需时间，对项目团队必需的人数规划没有什么帮助。一般来讲，在项目刚开始的时候只需要很少几个人，他们负责项目的初始设计。在项目开发和系统测试的时候，项目成员数达到最高峰。当系统完成准备部署的时候，人员数目开始下降。这就显示了项目人员的组织和项目时间的减少的关联关系。因此，项目管理者在项目整个生存期中都要避免过早把太多人员加到项目中来。

工作量分布可以用 Rayleigh 曲线（Londeix, 1987）建模，Putnam 的估算模型（Putnam, 1978）中的项目人员配备模型就是基于该曲线的。Putnam 模型还把开发时间作为一个关键因素，随着开发时间的减少，系统开发所需要的工作量呈指数增长。

要点

- 系统报价并不仅仅取决于它的估计开发成本和开发公司要求的利润。机构因素可能提高售价以补偿升高的风险，或者降低售价以获得竞争的优势。
- 软件通常是先有定价以得到合同，然后再据此调整相应的功能。
- 计划驱动开发是围绕一个详细定义的项目计划进行组织的。项目计划定义了项目活动、计划的工作量、活动进度安排和每项活动的负责人。
- 项目进度安排需要创建有关项目计划的各种图形化表示。用来表示活动期限和人员使用时间的条形图是在进度安排中最常使用的。
- 项目里程碑是一个项目活动可以预期的结果，到达一个里程碑就要把某些项目进展报告提交到管理层。在一个软件项目中，里程碑的出现应该是有规律的。可交付的文档则是交付到客户手中的里程碑。
- 极限规划游戏是让整个团队成员都参与到项目规划中来。计划做成增量式的，如果问题出现，计划将被调整，通过减少软件的功能性而不是延期交付一个增量，修改计划。
- 软件的估算技术可能是基于经验的，管理者对需要的工作量进行判断；或者是使用算法，需要的工作量通过使用其他的估算工程中的参数计算得到。
- COCOMO II 成本计算模型是一个比较成熟的成本估算模型，它将项目、产品、硬件以及人的因素都考虑在内。

进一步阅读材料

《Software Cost Estimation with COCOMO II》这是关于 COCOMO II 模型的权威书籍，提供了模型的完整的描述，有着大量的例子，而且包括了实现模型的软件。它非常详细但不易读懂 (B. Boehm, et al., Prentive Hall, 2000)。

《Ten unmyths of project estimation》这是一篇谈项目估算中实际困难并挑战这一领域一些基本假定的实用文章 (P. Armour, Comm. ACM, 45 (11), November 2002)。

《Agile Estimationg and Planning》这本书全面介绍如在 XP 中使用的基于情景的规划，同样也阐述了在项目规划中使用敏捷方法的基本原理。它也包括一个很好很的关于项目规划问题的概括 (M. Cohn, Prentice Hall, 2005)。

《Achievements and Challenges in Cocomo-based Software Resource Estimation》这篇文章展示了 COCOMO 模型的一个发展历史过程，及其在这些模型上产生的影响。讨论了这些模型的多种变化。它也介绍了 COCOMO 方法未来可能的发展前景。(B. W. Boehm and R. Valeridi, IEEE Software, 25 (5), September/October 2008)。http://dx. doi. org/10. 1109/MS. 2008. 133。

练习

- 23.1 在什么情况下公司可能会将软件系统的价格定得比成本估价加上正常利润高得多呢？
- 23.2 解释项目规划过程为什么是一个迭代的过程，为什么在一个软件项目期间必须不断地对项目规划进行评审。
- 23.3 简要介绍软件项目计划中每个部分的目的。
- 23.4 无论使用什么估算方法，成本估算有其固有的风险。试给出 4 个能降低成本估算风险的方法。
- 23.5 图 23-14 列出了许多活动、持续时间和各活动之间的依赖关系。请画出活动图和条形图示意项目进度安排。
- 23.6 图 23-14 给出了软件项目活动的任务持续时间。假设发生了一个严重的、意想不到的事件，使得任

务 T5 不是在 10 天内完成,而是用了 40 天。画出新的条形图以表示如何重新进行组织。

- 23.7 XP 规划游戏是基于规划的概念去实现表现系统需求的情景的。解释当软件有高性能和高可靠性要求时这个方法可能存在的问题。
- 23.8 有一个软件负责人负责一个安全性要求极高的软件系统的开发,该系统的设计是为了控制针对癌症患者的放射治疗仪。这个系统是嵌入式系统,运行于一种专用处理器,内存被限定在 256 MB。放射治疗仪与患者数据库通信以获取患者的详细资料,并在治疗完毕后自动地将放射剂量和其他详细治疗信息记录到数据库中。
- 在对系统开发的工作量估算中使用了 COCOMO 方法,计算结果是需要 26 人月。在估算中所有的成本形成因素都被设为 1。
- 解释为什么需要对这个估算进行修正,将项目、人员、产品和机构因素统统考虑在内。试列举出在初始 COCOMO 估算中会产生重要影响的 4 个因素,并对这些因素给出可能的取值。对于为什么考虑到这些因素给出合理的解释。
- 23.9 一些非常大型的软件项目都有数以百万行的程序代码行。解释为什么像 COCOMO 这样估算模型的工作量对非常大型的软件系统可能无效。
- 23.10 公司了解到客户需求不明确,在签订合同时故意提出一个低报价,待客户将来提出需求变更时再索要高价。你认为这样做道德吗?

任 务	工期 (天数)	依 赖 关 系
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T2, T14
T16	10	T15

图 23-14 进度安排实例

参考书目

Beck, K. (2000). *extreme Programming Explained*. Reading, Mass.: Addison-Wesley.

Boehm, B. 2000. 'COCOMO II Model Definition Manual'. Center for Software Engineering, University of Southern California. http://csse.usc.edu/csse/research/COCOMOII/cocomo2000.o/CII_modelman2000.o.pdf.

Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R. and Selby, R. (1995). 'Cost models for future life cycle processes: COCOMO 2'. *Annals of Software Engineering*, 1 57-94.

- Boehm, B. and Royce, W. (1989). 'Ada COCOMO and the Ada Process Model'. *Proc. 5th COCOMO Users' Group Meeting*, Pittsburgh: Software Engineering Institute.
- Boehm, B. W. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall.
- Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. and Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Upper Saddle River, NJ: Prentice Hall.
- Londeix, B. (1987). *Cost Estimation for Software Development*. Wokingham: Addison-Wesley.
- Myers, W. (1989). 'Allow Plenty of Time for Large-Scale Software'. *IEEE Software*, 6 (4), 92–9.
- Putnam, L. H. (1978). 'A General Empirical Solution to the Macro Software Sizing and Estimating Problem'. *IEEE Trans. on Software Engineering.*, SE-4 (3), 345–61.
- Schwaber, K. (2004). *Agile Project Management with Scrum*. Seattle: Microsoft Press.

质量管理

目标

本章旨在介绍软件质量管理，讲述专门的质量管理活动。读完本章，你将了解以下内容：

- 了解质量管理过程，以及质量规划重要的原因；
- 了解软件质量受到所使用的软件开发过程的影响；
- 认识到质量管理过程中质量标准的重要性以及标准是如何在质量保证中使用的；
- 了解复查和审查是如何作为机制在软件质量保证中使用的；
- 理解度量如何在评估某些质量属性时发挥作用，以及目前软件度量的局限性。

软件质量问题最早在 20 世纪 60 年代首次大型软件系统开发时期被发现，并且在整个 20 世纪一直困扰着软件工程。已交付的软件速度很慢而且不可靠，维护困难重重，并且还很难复用。由于对这种状况不满，人们开始采用软件质量管理形式化技术，这种技术是由制造业中的方法发展而来的。这些质量管理技术和新的软件技术以及更好的软件测试结合在一起，使得软件质量的一般水平得到了明显提升。

软件系统的软件质量管理有 3 个重要的关注点：

① 在机构层面，质量管理与建立能生产高质量软件的机构过程框架和标准相关。这就意味着质量管理团队应该对定义要使用的软件开发过程、软件采用的标准以及包括系统需求、设计以及代码的相关文档负责。

② 在项目层面，质量管理包括专门的质量过程的应用、对所规划的过程的执行情况的检查，及确保项目的输出符合此项目所适用的标准。

③ 在项目层面的质量管理同样关注于为项目确立一个质量计划。质量计划应该给出项目的质量目标，定义应该使用什么样的过程 and 标准。

制造业中广泛使用“质量保证”和“质量控制”这两个术语。质量保证（QA）是对生产高质量的过程和标准的定义，同时也引入质量过程到制造过程。质量控制是应用这些质量过程淘汰没有达到要求的质量水平的产品。

在软件产业中，不同的公司和工业部门以不同的方式解释质量保证和质量控制。有时，质量保证仅代表对旨在保证软件质量达标的流程、过程和标准的定义。另外的情况下，质量保证也包括开发团队交付产品后采取的活动，包括所有的配置管理、检验和有效性验证活动。在这一章中使用的“质量保证”包括：检验、有效性检验以及检查质量流程是否正确应用的过程。由于软件产业并未广泛使用“质量控制”这个术语，本书也将避免使用此术语。

质量保证团队（QA team）在大多数公司中负责管理版本测试过程。正如第 8 章中所讨论的，这就意味着在软件版本交付到客户手里之前，他们负责软件的测试。他们负责检查系统是否满足需求，以及维护测试过程记录。第 8 章中已经涉及版本测试，本章不再涉及有关这方面的质量保证问题。

质量管理对软件开发过程提供独立的检查活动。从软件过程中产生的可交付文档要放到质量管理过程中接受检验，确保它们能够符合机构的标准和目标（见图 24-1）。由于质量保证团队应该是独立于开发队伍的团队，他们能够客观地对待软件产品。这样他们就能够不受软件开发

问题的影响，做出客观的软件质量报告。

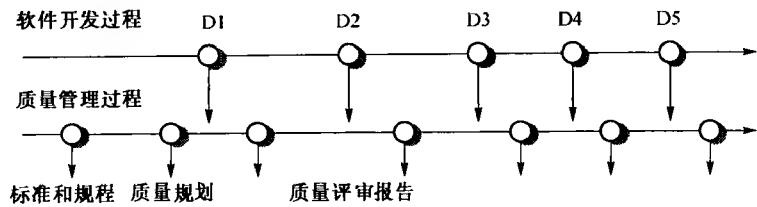


图 24-1 质量管理和软件开发

理论上说，质量管理团队应该不与任何专门的开发小组有关联，但是应对整个机构的质量管理负责。他们应当是独立的，并且直接向项目管理人员之上的管理者报告。这是因为项目管理人员必须维护工程预算开支和进度安排。如果出现问题，他们可能试图在产品质量上做出妥协来满足他们的工程进度。一个独立的质量管理团队确保机构的质量目标不会因为考虑短期的开支和进度而有所妥协。但是，在小一点的公司里，这是不太实际的。质量管理和软件开发不可避免地绑定在一起，相关的人员同时有着开发和质量两方面的职责。

质量规划是为项目制定一个质量计划的过程。质量计划应当列出要达到的软件质量，并且描述怎样评估这些质量。因此对于一个特定系统，计划定义何为高质量的软件。没有这样的定义，在关于哪种产品属性能反映出最重要的质量特点方面，工程师可能做出不同的、有时甚至是冲突的假设。正式的质量规划是基于计划的开发过程一个重要组成部分。然而，敏捷方法采用一种不太正式的方法来进行质量管理。

Humphrey（1989），在他的关于软件管理的经典书籍中，提出了一个质量规划的轮廓结构。其中包括：

1. **产品介绍** 说明产品、产品的意向市场及对产品性质的预期。
2. **产品计划** 包括产品确切的发布日期、产品责任以及产品的销售和售后服务计划。
3. **过程描述** 产品的开发和管理中应该采用的开发和服务过程 and 标准。
4. **质量目标** 产品的质量目标和计划，包括识别和判定产品的关键质量属性。
5. **风险和风险管理** 说明影响产品质量的主要风险和这些风险的应对措施。

质量规划被作为一般项目规划过程的一部分，依据所开发的系统的大小和类型而有所不同。但是，在书写质量计划时，我们应该尽力保证它们尽可能简短。如果文档过长，人们就不会去阅读，这样就会导致设定质量规划的初衷失败。

一些人认为软件质量能够通过检查过程来达到。机构会有一个基于机构标准的和按照相关质量流程的一个确定的检查过程，去检查软件开发团队在开发中是否遵循了这些质量标准。他们的观点是标准应该包含好的软件工程实践；遵从这个好的实践必然得到高质量的产品。然而，实际上质量管理的内容远比一些标准以及相关的保证这些标准得到执行的行政制度要多得多。

标准和过程固然是很重要的，但是质量管理者也应该致力于开发一种“质量文化”，让每个参与产品开发的人都有强烈的产品质量意识。他们鼓励团队对自己的工作质量负责，鼓励他们探求改善质量的新方法。尽管质量标准和规程是质量管理的基础，好的质量管理者还是认识到有些软件质量特性不易量化（如简洁性、可读性等），难以在标准中具体体现出来。他们支持在无形质量方面有兴趣的人并鼓励所有的团队成员的好的工作作风。

正式的质量管理对于开发大型、长期（开发需几年时间）的系统的团队而言特别重要。质量文档是记录项目中的每个子小组所做的工作的文件。它帮助检查以避免遗忘重要的任务或团

队的一部分不会对其他团队所做的工作做出错误的假设。质量文档也是贯穿一个系统生命周期的沟通手段。它允许对系统进化负责的小组追踪测试以及开发团队所做的工作。

对较小的系统，质量管理也是重要的，但可以采用相对非正规的方法。由于小的开发团队可以随意的交流，所以不必需要那么多的文书工作。对于小系统的开发，关键的质量问题是建立质量文化并保证所有的团队成员对软件质量有一个有效的方法。

24.1 软件质量

质量管理的基本原则是制造工业为了改善制造产品的质量而建立的。作为质量管理的一部分，他们首先定义了什么是“质量”。所谓质量是基于详细产品描述的（Crosby, 1979）和公差概念的。基本假设是，能够完整详细地定义产品，并且能够确立一个依照产品描述检查制造产品的流程。当然，产品不会完全精确地满足描述，所以一定的公差是可以允许的。如果产品是“基本合适的”，那么它就被认为是合格的。

软件质量不能直接和制造业中的质量相比较。公差思想对于数字系统来说是不适用的。由于以下原因，可能没法得出关于软件系统是否满足描述的客观结论：

1. 正如第4章所讲的需求工程，对于写出一个完整的和无歧义的软件描述是相当困难的。软件开发商和客户可能对于需求有不同的阐述，并且可能对软件是否符合描述没法达成共识。

2. 描述通常整合了各类信息持有者的需求。这些需求不可避免地存在着取舍，很可能没有包含所有类别信息持有者的需求。排除在外的信息持有者可能认为系统质量糟糕，即使它实现了那些达成共识的需求。

3. 对某些质量特性（如，可维护性）的度量是不可能做到的，所以它们是不能以一种无歧义的方式描述的。24.4节讨论了度量的困难性。

因为以上这些原因，评估软件质量是一个主观的过程，质量管理团队必须判断决定软件是否达到可接受的质量水平。质量管理团队必须考虑软件是否达到既定的目标。这涉及回答关于系统特性的若干问题。例如：

1. 开发过程是否遵循编程和文档化标准？
2. 软件是否得到了正确的测试？
3. 软件是否足够可靠能被投入使用？
4. 软件性能是否对于正常使用是合格的？
5. 软件是否可用？
6. 软件是否结构良好并且易于理解？

软件质量管理存在一个通用的假设：按照需求测试系统。应该根据这些测试的结果判断是否实现了要求的功能。因此，质量保证（QA）团队应该复查所设计的测试并检查测试记录以核实测试是否被正确地执行。在一些机构中，软件管理团队负责系统测试。但是有时，一个独立的系统测试小组负责测试。

一个软件系统的主观质量很大部分依赖于它的非功能性特性。这反映了实际的用户体验——如果软件的功能不是所期望的那样，那么用户就会变通，寻找其他方式来做他们想做的事情。但是如果软件不可靠或者是速度太慢，那么实际上就不能达到他们的目的。

因此，软件质量不仅仅取决于软件功能是否正确地实现，也取决于非功能的系统属性。Boehm等人（1978）指出有15种重要的软件质量属性，如图24-2所示。这些属性和软件可靠性、可用性、有效性以及可维护性相关。

安全性	可理解性	可移植性
信息安全性	可测试性	可用性
可靠性	可调节性	可复用性
适应力	模块化	效率
鲁棒性	复杂度	学习能力

图 24-2 软件质量属性

如第 11 章中所讲述的，可靠性属性通常是最重要的系统质量属性。但是软件性能也很重要。如果软件太慢，用户会拒绝使用。

对于任何系统，优化所有属性是不太可能的，例如，提升鲁棒性可能导致性能的降低。因此质量规划应该定义被开发软件最重要的质量属性。可能有效性很关键，可能需要牺牲掉其他属性。如果已经在质量规划中声明了这一点，从事开发工作的工程师会协力来达到这一目标。计划同样应该包括定义质量评估过程。这应该是各方都认可的评估，判断是否在产品中存在一些质量属性，如可维护性和鲁棒性。

质量管理的一个基本假定是开发过程的质量直接影响产品的质量。这个假定源于生产制造系统中产品质量与生产过程的密切关系。制造过程包括配置、工装夹具准备和操作相应的机器。一旦机器工作正常，产品质量自然就有保证。评估产品质量并改变生产过程直到达到需要的质量水平。图 24-3 表现了这个基于过程得到产品质量的方法。

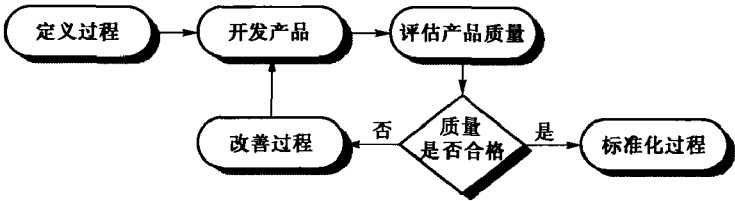


图 24-3 基于过程的质量

在生产制造中过程与产品质量有着明确的关联，因为过程相对易于标准化和监控。一旦生产系统校准后就能一次次生产出高质量的产品。而软件不是生产制造出来的而是设计出来的。因此，软件开发过程中过程质量和产品质量之间的关系更加复杂。软件开发是创造性活动而不是一个机械过程，个人的技能和经验的影响非常大。无论所使用的过程怎样，一些外部因素（如，应用程序的创新性或早期产品发布的商业压力）也会影响产品质量。

毫无疑问，使用的开发过程对于软件质量有明显的影响。好的过程更有可能得到好质量的软件。过程质量的管理和改进能够减少软件开发过程中产生的缺陷。但是，评估软件质量的属性非常困难，比如，不经过长时间使用软件，很难评估可维护性。因此，很难指出过程特性如何影响这些属性。此外，因为设计和创造性在软件过程中所起的作用，过程标准化有时会扼杀创造力，这会导致软件质量更糟而不是更好。

24.2 软件标准

软件标准在软件质量管理中扮演着重要的角色。正如前文所述，质量保证一个重要部分是定义和选择应用于软件开发过程和软件产品的标准。作为质量保证过程的一部分，也要选择支持标准使用的工具和方法。一旦选定使用标准，必须定义项目特定的过程以监控标准的使用和执行情况。

- 软件标准非常重要，有 3 个原因：
1. 标准是智慧的结晶，对一个机构有重要意义。软件标准封装了对于机构来说最成功的或是最恰当的软件开发实践。这些知识往往是经过反复实验和无数的挫折后才得出的。把这些知识制定到标准中去可以避免重犯过去的错误。
 2. 标准为定义特定环境中的“质量”含义提供了一个框架。如前文所述，软件质量是主观的，通过使用标准，为判断软件是否达到要求的质量水平建立基础。当然，这依赖于反映用户对软件可靠性、可用性以及性能的期望的环境标准。

3. 软件标准还有助于工作的连贯性，由一个人着手进行的工作别人可以接着做。软件标准确保一个机构中所有的工程人员采用相同的做法。这样一来，开始一项新工作时就节省了学习时间。



文档化标准

项目文档是一种看得见摸得着的描述软件系统以及生产过程的不同形态的方式（需求、UML、代码等）。文档化标准定义不同类型文档的组成以及文档的格式。这是很重要的，因为这样可以很容易发现是否有重要的内容被遗漏，并确保项目文档有一个普遍接受的外观。标准会针对书写文档的过程、文档本身内容以及文档交换诸多方面分别制定。

<http://www.SoftwareEngineering-9.com/Web/QualityMan/docstandards.html>

在软件质量管理中，现存两类可用于定义和使用的相关软件工程标准：

1. **产品标准** 这些标准用于开发的软件产品。包括文档标准，如生成的需求文档的结构；文档编写标准，如定义对象类时注释标题的标准写法；还有编码标准，它规定如何使用某种程序语言。

2. **过程标准** 这些标准定义了软件开发必须遵循的过程。应将良好的开发方法封装其中。过程标准包括对描述、设计和有效性验证过程、过程支持工具以及对在这些过程中产生的文档的描述的定义。

标准必须以提升的产品质量的形式表现价值。有的标准需要花费大量时间和劳动，但是只是带来了细微的质量改进，这种标准是没有必要定义的。必须设计产品标准，然后可以应用和以有效的方式检查标准，过程标准应该包含用于检查是否遵循产品标准的过程定义。

国际软件工程标准的发展通常是一个持续很久的过程，那些对标准感兴趣的人聚集在一起，然后起草评论，最终对标准达成一致。一些国家和国际组织，如美国 DOD、ANSI、BSI、NATO 和 IEEE，都支持标准的制定工作。这些制定出来的标准具有普遍性，能够适用于许多领域内的项目。像 NATO 和其他的国防机构就需要在他们和软件公司的软件开发合同中应用自己使用的执行标准。

已经制定的国家标准和国际标准涵盖了软件工程技术语、编程语言（如 Java 和 C++）、符号系统（如制图符号）、软件需求的导出和书写规程、质量保证规程以及软件检验和有效性验证过程（IEEE，2003）等许多方面。更多的专属标准，比如说 IEC 61508（IEC，1998），是为了安全性和信息安全性要求极高的系统而开发的。

质量管理团队在制定机构标准时，一般要参照国家标准和国际标准。以这些标准作为出发点，质量保证团队应该拟定一本标准“手册”，定义适合自己机构的标准。这种手册可能要包含的标准种类列于图 24-4 中。

软件工程人员有时会把软件标准视为一种行政命令，是与软件开发的技术活动毫不相干的，尤其是在标准中要求填写烦琐的表格和工作记录的时候。尽管他们大都承认贯彻实施通用标准是十分必要的，但工程师们总能找出一些理由，力图说明某些标准并不适合他们的具体项目。为了尽量减少不满意情绪，因此设定这些标准的质量管理人员要采取以下步骤：

1. **让软件工程人员参与产品标准的选择** 如果开

产品标准	过程标准
设计复查表	设计复查方式
需求文档结构	为系统构建提交新代码
方法头格式	版本发布过程
Java 编程风格	项目计划批准过程
项目计划格式	变更控制过程
变更请求表	测试记录过程

图 24-4 产品和过程标准

发者了解了所选择的标准的原因，就会自觉执行这些标准。最理想的情况，标准文档不应只是列出需要执行的标准，还应该包括评论性的解释，说明为什么得出这样的标准化的决议。

2. 定期评审和修改标准，以反映技术的变化 开发标准代价不菲，标准一经制定出来就要载入公司的标准手册。由于成本和所需要的讨论，通常不会轻易对标准进行改动。标准手册是必备的，但是它要随着环境和技术的变化而不断完善。

3. 尽可能提供支持软件标准的软件工具 保持标准的一致性包括乏味的手工工作，而这些工作是可以由软件工具完成的，开发人员经常觉得标准是大麻烦。如果有工具支持，遵循软件开发标准就只需要很少的成本。例如文档标准可以通过使用文字处理器样式实现

不同类型软件需要不同的开发过程，所以必须采用适当标准。如果某种工作方式不适合一个项目或项目团队，对它做出规定是没有意义的。因此每个项目管理者都应该有根据个别情况改动标准的权力。然而，当做出变更时，保证这些变更不会影响产品质量是很重要的。这会影响到一个机构和它的顾客之间的关系，并且很可能导致项目成本的上升。

项目管理者和质量管理者可以通过切实可行的质量规划避免标准的不适当问题。他们应该确定质量手册中哪些标准应该不折不扣地执行，哪些标准应该修改，哪些标准应该废止。对于某些用户和特定的项目需求可以制定相应的标准。例如，如果以前的项目中没有用到形式化描述的标准，就需要制定这些标准。

ISO 9001 标准框架

ISO 9000 是一个用于在所有行业建立质量管理体系的国际标准集。ISO 9000 可应用的范围很广，从制造业到服务业都有涉及。ISO 9001 在这些标准中是最具普遍性的标准，它适用于设计、开发和产品维护等机构内的质量过程，包括软件。ISO 9001 标准最初开发于 1987 年，它的最新版本是 2008 年发布的。

ISO 9001 标准自身并不是软件开发的一个标准，而是开发软件标准的一个框架。它制定出一般的质量原则，描述一般的质量过程，并且编排应该定义的组织标准和步骤。这些应当记录在机构质量手册中。

ISO 9001 标准在 2000 年进行了一次重大的修订，形成了 9 个核心过程，如图 24-5 所示。为了服从 ISO 9001 标准，公司必须记录它们的过程是如何与这 9 个核心过程相对应的。也必须定义和维护有关记录证明所定义的机构过程已经得到了严格执行。公司的质量手册应该描述相关的过程以及过程数据，这些数据必须收集并得到维护。

ISO 9001 标准并没有定义或规定公司中应该使用的特定的质量过程。要与该标准一致，公司必须定义过程的类型，如图 24-5，并有相应的流程证明它的质量过程是得到严格遵守的。这就带来了不同产业部门和公司规模之间的灵活性。这就能够定义适合所开发的软件类型的质量标准。小型公司能够拥有灵活的但同时仍然是服从 ISO 9001 标准的过程。但是，这种灵活性意味着我们不能对于遵循 ISO 9001 标准的不同公司过程之间相似性和差异性作出假定。一些公司可能拥有很严格的质量过程来保留详细的记录，然而其他一些公司可能不那么正式，只有极少量的附加文档。



图 24-5 ISO 9001 核心过程

ISO 9001，机构的质量手册和个体的项目质量计划之间的关系如图 24-6 所示。这个图源于

由 Ice 提出的一个模型 (1994)，他解释了通用的 ISO 9001 标准如何作为一个软件质量管理过程的基础来使用。Bamford 和 Dielbler (2003) 解释了后来的 ISO 9001:2000 标准如何在软件公司中被采用。

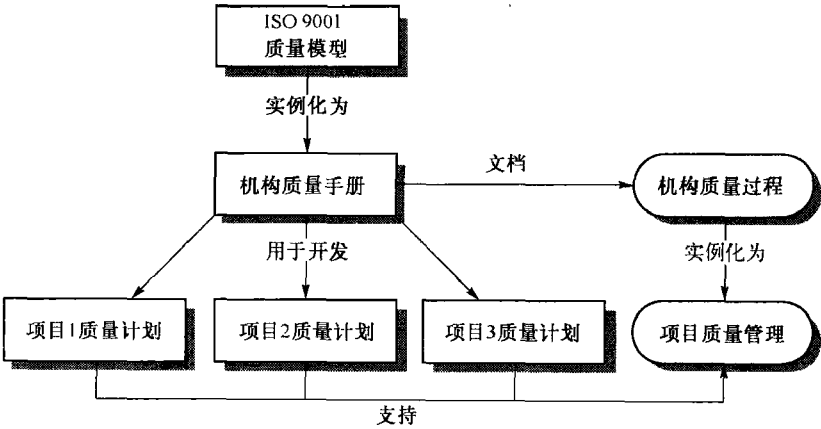


图 24-6 ISO 9001 和质量管理的流程图

一些软件客户要求他们的供应商是具有 ISO 9001 认证。软件开发公司拥有质量管理体系的认证，客户才会有信心。拥有独立鉴定资格的机构对质量管理过程和过程文档进行检查，判断这些过程是否符合 ISO 9001 标准所有内容。如果的确如此，就像质量手册里定义的一样，他们认证这家公司的质量过程符合 ISO 9001 标准。

有些人认为 ISO 9001 证书意味着经过认证的公司所生产的软件质量比未经认证公司要好，这不一定。ISO 9001 标准只是关心机构里拥有质量管理过程，并且遵循这些过程。并没有保证说 ISO 9001 认证的公司使用最好的软件开发实践，或是这些过程会产生高质量的软件。

比如，某家公司所定义的测试覆盖标准规定、对象中所有方法必须使用至少一次。不幸的是，这个标准会在不完全的软件测试中使用，对使用不同方法的参数并没有进行测试。只要遵循了定义的测试流程，并且记录了执行的测试过程，这个公司就是满足 ISO 9001 认证的。ISO 9001 认证所定义的质量是符合标准，并不是软件用户所能体验的质量。

敏捷方法中避免使用文档，它只关注于开发的代码，与 ISO 9001 中所讨论的形式化的质量过程差别甚大。在使这两种方法趋为一致的方面，已经有一些人做过一些工作 (Stalhane 和 Hanssen, 2008)，但是敏捷开发群体本质上是完全反对这种他们视为官僚作风的标准化。由于这样的原因，使用敏捷开发方法的公司很少关心 ISO 9001 标准认证。

24.3 复查与审查

复查 (review) 与审查 (inspection) 是检查项目可交付文档的质量的 QA 活动。检查的内容涉及：检查软件，软件文档，以及发现错误和遗漏的过程的记录，并且检查是否遵循质量标准。第 8 章和第 15 章提到，复查、审查和程序测试作为软件检验和有效性验证通用过程的一部分。

在复查过程中，一个团队检查软件与其相关文档，寻找潜在问题和与标准不一致的部分。复查团队提供资料来判断系统的质量级别和项目的可交付物。然后项目管理人员使用这些评定来做出规划决策并为开发过程分配资源。

质量复查基于在软件开发中产生的文档来进行。软件描述、设计、代码、过程模型、测试计划、配置管理规程、过程标准以及用户指南也都被复查。复查应当检查文档和代码的一致性和完整性，确保遵循质量标准。

然而，复查不仅仅是检查与标准的一致性，还被用来帮助发现软件和项目文档中的问题和遗漏。复查的结果应当作为质量管理过程的一部分被正式记录。如果发现了问题，应将复查人员的意见交给开发者或者负责修改所发现问题的人员。

复查和审查的目的是提升软件的质量，不是评估开发团队中员工的表现。相对于较为私下进行的组件测试过程，复查是一个检测错误的公开过程。不可避免的是，个人犯的错误会暴露在整个编程团队面前。要确保所有开发人员对复查过程起到有建设性的作用，项目管理人员必须对个人的关注保持敏感。他们必须营造一种工作文化，发现错误时不责备当事人。

尽管质量复查为管理提供关于软件开发的信息，但是质量复查不同于管理过程复查。第 23 章中提到，过程复查将软件工程的实际过程与计划的过程对比。他们主要的关注点是工程是否能够按时并在预算范围内发布有用的软件。过程复查将外部因素考虑在内，环境变化可能导致不再需要开发的软件或是必须作出彻底改动。由于业务或它的操作环境发生了改变，导致已开发出高质量软件的工程不得被撤销。

24.3.1 复查过程

尽管在复查的细节上有很多不同，但是复查过程（见图 24-7）一般分为 3 个阶段：

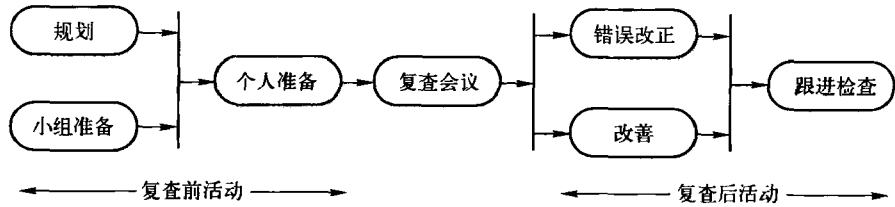


图 24-7 软件复查过程

1. **复查前活动** 这些准备工作对于复查的有效进行是必须的。具有代表性的是，这些复查前工作关心复查的计划和复查的准备工作。复查计划包括建立一个复查团队，安排复查的时间地点，分发要被复查的文档。在复查准备工作中，复查团队见到要复查的软件的一个综述。个别的复查团队成员需要阅读并理解软件、文档以及相关的标准。他们独立地工作，依靠标准找出错误、遗漏和违背的地方。复查人员如果不能参加复查会议，他们可以提供书面的软件意见。

2. **复查会议** 在复查会议期间，被复查文档或程序的作者应该和复查团队一起把文档从头到尾浏览一遍。复查本身时间应该是相对短的，至多两个小时。一个团队成员应该作为复查的主席，还应该有一个成员正式地记录所有复查决议和要采取的行动。在复查期间，主席负责保证所有的书面意见都被考虑在内。复查主席应该在复查期间写下一个达成共识的意见和行动的记录。

3. **复查后活动** 在复查会议结束后，必须解决在复查期间提出的问题。这可能包括修复软件漏洞，重构软件以使它与质量标准相一致，或是重写文档。有时，在质量复查中发现的问题要求管理复查也必须决定是否加入更多资源来改正它们。在做出改动之后，复查主席会检查所有被考虑的复查意见。有时，要求采取进一步的复查来检查覆盖了所有之前的复查意见而做出的改动。

复查团队应该挑选 3~4 名主要复查员作为团队的核心，应该有一个资深设计人员负责做出技术上的重大决策。主要复查员可以邀请其他的项目成员帮助复查，他们不必参与整个文档的复查，而应集中精力解决影响他们工作的问题。另外，复查团队可以把要复查的文档进行传阅，并要求其他的项目成员写出书面意见。项目管理人员不需要包括到复查中，除非预见的问题要求改变项目计划。

上述的复查过程需要开发团队中所有成员都位于同一地点，并且可以参加团队会议。但是，现在工程团队通常是分散的，有时候分布在几个不同国家甚至大洲，所以团队成员聚到一个屋子里开会通常是不实际的。在这种情形下，可以使用文档编辑工具支持复查过程。团队成员使用这些来评论文档和软件源代码。这些评论对于其他团队成员是可见的，他们可以赞成或者反对。只有在复查人员之间的分歧解决之后，才会使用电话讨论。

敏捷软件开发中的复查过程通常是非正式的。例如，在 Scrum 中，在每次的软件迭代完成后都会有一个复查会议（冲刺复查），其中会讨论到质量问题。在极限编程中（下一节中讨论），配对编程确保另一个团队成员经常检查和复查代码。日常的团队会议中会考虑到通常的质量问题，但是极限编程依赖于个人主动性来提升和重构代码。通常敏捷方法不是标准驱动的，所以通常不考虑标准一致性问题。

在敏捷方法中缺少正式的质量过程，这意味着已经开发详细的质量管理过程的公司使用敏捷方法可能存在问题。质量复查可能减慢软件开发步伐，但在计划驱动的开发过程中质量复查能得到最好的使用。在计划驱动的过程中，规划复查，并与其他调度工作平行进行。在敏捷方法中因专注于代码开发，所以这样做是不现实的。



审查过程的角色

当 IBM 首次建立程序审查的时候 (Fagan, 1976; Fagan, 1986)，对审查小组的成员有多个正式的角色分工。这些角色包括：协调员，代码阅读者，抄写员。审查过程的其他用户修改了这些角色，但是通常都接受审查中要有：代码作者，一个督察员，一个抄写员，由协调员来主持。

<http://www.SoftwareEngineering-9.com/Web/QualityMan/roles.html>

24.3.2 程序审查

程序审查是“同行评审”，团队成员合作来发现开发程序中的漏洞。第8章中谈到，审查可以作为软件检验和有效性验证过程的一部分。因为它们不要求执行程序，所以它们和测试互补。这就表示能够验证系统不完整的版本，并且能够检查像统一建模语言（UML）模型这样的表示法。Gilb 和 Graham (1993) 提出了一种使用审查最有效的方法，那就是复查系统测试用例。审查能够发现测试的问题，并且提升这些测试检测程序错误的有效性。

程序审查涉及来自不同背景的团队成員，他们对程序源代码进行精心的、一行一行的复查。他们寻找错误和问题，并且在审查会议中描述出来。错误可能是逻辑错误，也可能是代码中的异常，这些异常可能表明了错误情况或者代码忽略的特征。复查团队详细检查设计模型和程序代码，并且标记出需修正的异常和问题。

审查时，经常使用一份常见编程错误的检查表。这份检查表基于的是来自书本的实例和个别应用领域的错误经验。我们对于不同的编程语言使用不同的检查表，因为每种语言有它自己特有的错误。Humphrey (1989) 在对审查的详细讨论中，给出了多个检查表的例子。

在审查过程中可能做出的检查如图 24-8 所示。Gilb 和 Graham (1993) 强调每个机构都应当根据部门标准和实践开发自己的检查表。由于不断发现新的错误类型，这些检查表应经常更新。因为编译期间会有不同的检查级别，检查表的条目根据编程语言不同而有区别。例如，Java 编译器检查函数的参数个数是否正确，但是 C 编译器却不检查。

大多引入审查的公司发现它们在发现漏洞方面是非常有效的。Fagan (1986) 报告了使用非

形式化的程序审查可以检测到 60% 以上的程序错误。Mills 等人（1987）提出了一个更加形式化的检查方法，根据正确性论证，能够检测出 90% 以上的程序错误。McConnell（2004）对比了单元测试，其错误探测率大约为 25%。通过审查，错误探测率达到了 60%。他同样描述了许多案例研究，包括引入“同行评审”的例子，它会使生产率提升 14%，同时程序错误降低 90%。

尽管他们大量宣传其成本有效性，很多软件开发公司还是不愿意使用审查或者同行评审。有程序测试经验的软件工程师有时会不愿意接受关于审查会比测试在发现错误方面更加有效的观点。管理人员也可能产生怀疑，因为在设计和开发过程中审查工作要求额外的开销。他们不希望冒风险，即在程序测试中没有节省相应的成本。

缺陷分类	检查内容
数据缺陷	所有的程序变量都在使用前被初始化了吗？ 所有的常数都命名了吗？数组的上边界应该等于数组长度还是长度减 1？如果使用字符串，定界符应该显式的指定吗？有缓冲区溢出的可能性吗？
控制缺陷	对每一个条件语句，条件是正确的吗？ 每一个循环都能终止吗？ 复合语句被正确地扩起来了么？ 对 case 语句，所有可能的情况都考虑到了吗？若每一个 case 语句都需要跟一个 break 语句，有遗漏吗？
输入/输出缺陷	所有的输入变量都使用了吗？所有的输出变量在输出前都被赋值了吗？有未料到的输入引起系统崩溃吗？
接口缺陷	所有的函数和方法调用都使用了正确数量的参数吗？形参和实参类型匹配吗？参数顺序都对吗？如果组件访问共享内存，它们都有相同的共享内存结构模型吗？
存储管理缺陷	如果一个链接的结构被修改了，所有的链接都得到重新赋值了吗？如果使用了动态存储，空间分配正确吗？如果空间不再使用，需要显式地对空间释放吗？
异常管理缺陷	所有可能的错误状态都已经考虑到了吗？

图 24-8 审查过程中的检查表

敏捷过程很少使用形式化的审查和同行评审过程。但是，他们依赖于团队成员合作来检查每一个其他成员的代码，也信奉非正式的实用准则，比如说“提交前检查”，这表明程序员应该检查他们自己的代码。极限编程从业者认为配对编程是一种有效的检查方法，这实际上是一个连续的检查过程。两个人查看代码的每一行，并且在它被接受之前检查它。

配对编程会使人员对程序有一个深入的了解，因为两个程序员必须理解它的工作细节才能继续开发。这种了解的深度有时很难在其他检查过程中达到，因此配对编程能都找出正式检查过程有时都不能发现的漏洞。但是，配对编程也能够导致彼此之间的对于需求的误解，两个成员犯了同样的错误。此外，因为两人不想减缓工程的进度，两个人可能不愿意查找错误。相关人员不能跟外部检查团队那样客观，他们发现错误的能力很可能因为亲密的工作关系减弱。

24.4 软件度量和量度

软件度量（measurement）就是对软件组件、系统或过程的某种属性进行量化。在得到的数据之间以及数据和机构的通用标准之间进行比较，就可以得出有关软件质量或评估软件过程有效性、工具和方法有效性的结论。

举个例子，假设一个机构计划引入新的软件测试工具，在引入这个工具之前，记录下在一定时间内发现的软件缺陷数目。这是评估工具有效性的基础。使用工具一段时间后，重复这个过程。如果引入该工具后在相同的时间内发现的缺陷数目增多，就可以认为这种工具能给软件有效性验证过程提供有益的支持。

软件度量的长期目标是利用度量代替复查来对软件质量进行评判。使用一系列量度对软件进行度量能非常理想地评估一个系统，通过度量可以推断出系统的质量水平。如果一个软件达到了所需的质量阈值，那么它就可以不通过复查而被接受。适当的情况下，度量工具还可以突出显示出软件需要改进的部分。然而，现实距离理想情况还相距甚远。想要达到自动质量评估的理想状况在可预见的将来还不太可能。

软件量度 (metric) 是能够被客观度量的软件系统、系统文档或开发过程有关的特性。量度的例子包括：以代码行数表示的软件产品规模；雾 (Fog) 指数 (Gunning, 1962)，它是一段文本段落的可读性的一种指标；交付的软件产品中所报告的缺陷数；开发一个系统组件所需的人-日数等。

软件量度要么是控制量度要么是预言者量度。正如其名，控制量度支持过程管理而预言者量度帮助预测软件的特性。控制量度通常与软件过程相关。修复发现的缺陷所需平均工作量和时间是控制量度或过程量度的例子。预言者量度又叫产品量度，与软件本身相关，预言者量度的例子有：模块的回路复杂性 (第8章已讨论过)，程序中标识符的平均长度，在设计中与对象有关的属性和操作的数量。

无论控制量度还是预言者量度，都能影响管理决策的制定，如图24-9所示。管理者使用过程量度来决定是否做出过程改变，使用预言者量度来估计软件变更所需的成本。本章侧重于讨论预言者量度，通过分析软件系统的代码评估它的价值。第26章讨论控制量度及其在过程改进中的使用。

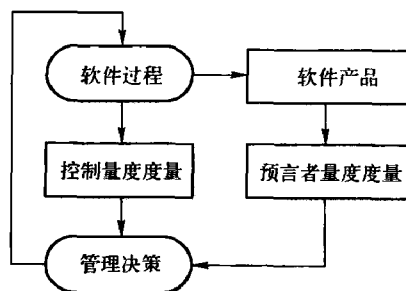


图 24-9 预言者度量和控制度量

软件产品度量可能用到两种方法：

1. **给系统质量属性赋值** 通过度量系统组件的特性，比如回路复杂性，并将这些度量综合起来，就能评估系统质量属性，比如可维护性。
2. **找出质量低于标准的系统组件** 度量能识别那些特性背离某些规范的个别组件。例如，可以度量组件以发现那些有着最高复杂性的问题组件。由于复杂度高难于理解，这些组件更可能包含错误。

不幸的是，就像图24-2所示，直接测试软件的质量属性是非常困难的。像可维护性、易懂性和可用性等质量属性是外部属性，与开发者和用户如何使用软件有关。它们受到主观因素的影响，比如用户的经验和知识使他们无法客观地度量软件。为了对这些因素做出判断，开发者不得不度量软件的某些内在属性（如软件的规模、复杂性等）并假定在所能度量的属性和想要了解的质量之间存在着一定的关系。

图24-10给出了某些可能是我们关心的外部软件质量属性和与其有关一些内在属性。该图说明了在外部和内部属性之间会存在某些关系的，但没有说明这些属性是如何关联的。内在属性的度量能否对外部的软件特性做出有益的预测，取决于以下3个条件 (Kitchenham, 1990)：

1. 内在属性必须被精确度量。这个往往并不是很简单的，它可能需要使用特殊工具来度量。
2. 在能够度量的属性和我们感兴趣的外部质量属性之间必须有一定关系。也就是说，在某种程度上质量属性的值必须和可度量属性的值相关联。
3. 内部属性和外部属性的关系必须是可理解的、可验证的、能用公式或模型表达出来。模型的公式化表示需要识别模型的函数形式（线性的、指数的等），这是通过分析收集到的数据，找出模型中要包含的参数，并用现有数据校正这些参数来完成的。

软件的内在属性，比如组件的环路复杂度，使用分析软件源代码的软件工具来度量。有开源工具可以使用来完成这些度量。虽然直觉上软件组件的复杂度和在应用中可观察到的错误数之

间有一定的联系，但是客观地用事实证明这些还是有一定的困难的。为了验证这种假设，开发者需要分析大量的组件失败数据和组件源代码。只有极少的公司对收集他们软件的数据做出长期的承诺，因此用来分析的失败数据是很难获取的。

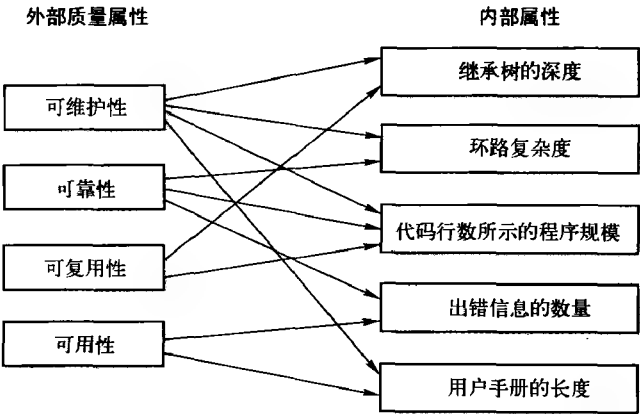


图 24-10 软件的内在和外在关系

20 世纪 90 年代，许多大型公司，像 Hewlett-Packard (Grady, 1993)、AT&T (Barnard 和 Price, 1994) 和 Nokia (Kilpi, 2001) 都引入了量度程序。他们对产品和过程进行度量，并在质量管理过程中使用这些度量。绝大多数工作焦点是有关程序缺陷、检验和有效性验证过程的量度的收集上。Offen 和 Jeffrey (1997)、Hall 和 Fenton (1997) 讨论了把量度程序引入到工业中的细节问题。

目前几乎没有公共可用的关于工业上系统地使用软件度量的信息。许多公司的确收集了软件信息，例如需求变更数或在测试中发现的缺陷数。然而，还不清楚是否它们接下来系统地使用了这些度量去比较软件产品和过程或是评估变更对软件过程和工具的影响。系统地度量比较困难的原因如下：

1. 引入一个机构量度程序的投资回报是无法量化计算的。在过去的几年里，软件的质量已经取得了极大的改善，这是在并没有使用量度的情况下取得的，因此很难判断引入系统的软件度量和评估的初始开销。
2. 现在还没有软件度量的标准，或是没有标准化的度量和分析过程。在这些标准和支持工具出现之前，大多数公司不愿意引入度量。
3. 在许多公司，软件过程是非标准化的，没有很好定义也是很难控制的。在某种意义上讲，在同一家公司内部以有效方式使用度量还存在很多过程变数。
4. 许多关于软件度量和量度的研究主要侧重于基于代码的量和计划驱动开发过程。然而，越来越多的软件的开发是通过配置 ERP 系统、COTS 系统或通过使用敏捷方法。因此，开发者们并不知道先前的研究是否适用于这些软件开发技术。
5. 引入度量增加了额外的开销。这和敏捷方法的目标相矛盾，敏捷方法推崇消除那些和程序开发没有直接关联的过程活动。因此已经采取的敏捷方法编程的公司并不倾向于采用度量程序。

软件度量和量度是经验软件工程的基础 (Endres 和 Rombach, 2003)。这是一个新的研究领域，它通过对软件工程的实验和对真实项目的数据的收集来建立和验证关于软件工程方法和技术的假设。致力于这个领域的研究者们认为，只有当我们能够提供具体的证据来证明某些软件工程方法和技术真正如它们的提出者所声称的那样带来效益，我们才能相信此软件工程的方法

和技术。

不幸的是，即使能够做出客观的度量并能从中得出结论，这些结论也未必令决策者们信服。与之相反，决策往往受到某些主观因素的影响，比如创新性，或是专业人员对该技术的感兴趣程度。因此，经验的软件工程结果要想对软件工程实践产生深远的影响仍需时日。

24.4.1 产品量度

产品量度是用来度量一个软件系统内在属性的预言者度量。产品量度的例子包括：系统大小，代码行数，每个对象类的方法数等。不幸的是，容易度量的软件特性如规模大小和环路复杂性，与如易懂性、可维护性等质量属性之间没有一个清晰而又一致的关系。这种关系是随着开发过程、技术以及被开发系统类型的不同而不同的。

产品量度分为以下两类：

1. 动态量度，通过对执行中的程序度量所收集到的。在系统测试期间或系统投入使用后可以收集到这些量度。例如出错报告的数量或完成计算所花费的时间。

2. 静态量度，通过对系统各种表现形式（如设计、程序或文档等）度量所收集到的。静态量度的例子有：代码多少和已使用标识符的平均长度。

这些量度类型与不同的质量属性有关。动态量度用于评估一个程序的效率和可靠性，而静态量度则用于评估一个软件系统或系统组件的复杂性、易懂性和可维护性。

动态量度与软件质量属性的关系通常较为密切。度量特定函数的执行时间和评估系统的启动时间相对比较容易，它们与系统的效率有直接的关系。同样，系统失败数和失败的类型要记录下来，它们直接关系到软件的可靠性，如第15章讨论的。

正如图24-11所示，静态量度与质量属性的关系是间接的。人们已提出很多这类量度，并进行试验，试图导出和验证这些量度与系统的复杂性、易懂性以及可维护性之间的关系。其中的程序长度和控制复杂性能对易懂性、系统复杂性和可维护性做出最可靠的预测。

软件量度	描 述
扇入/扇出	扇入是对调用其他函数或方法的函数数或方法数（假设用X表示）的度量。扇出是被X调用的函数数。一个高的扇入值意味着X与其他的设计紧密结合，对X的修改将产生广泛的影响。一个高的扇出值意味着X的整体复杂度可能很高，因为协调被调用组件所需的控制逻辑的复杂度
代码长度	这是对程序大小的度量。通常一个组件的代码越多，就越复杂并容易出错。代码长度是预测组件中易出错程度的最可靠的量度之一
环路复杂度	这是对程序控制复杂度的估量。这个控制复杂度可能与程序的易懂性有关。第8章介绍了怎样计算环路复杂度
标识符长度	这是对程序中标识符的平均长度的估量。标识符越长含义可能就越明确，程序也就越可理解
条件嵌套深度	这是程序中if条件嵌套深度的估量。较深的if条件嵌套难以理解并可能容易出错
Fog 指数	这是对文档中词和句子平均长度的估量。Fog指数的值越高，文档就越难以理解

图 24-11 静态软件产品量度

在图24-11中的量度是通用的，也提出了专门面向对象的量度。图24-12概括了Chidamber和Kemerer套件（有时称为CK套件）的6个面向对象的量度（1994）。虽然在20世纪90年代提出来了，但它们仍然是使用最为广泛的面向对象量度。当创建UML图时，一些UML的设计工具自动收集这些量度值。

El-Amam（2001）在对面向对象量度的一个非常好的评论中，讨论了CK量度和其他一些面

向对象的量度，并得出结论，我们还没有足够的证据来弄清楚面向对象量度是如何与外部软件质量关联的。直到 2001 年他分析之后，情况才真正改变。开发者们仍然不知道如何使用面向对象度量去得出有关软件质量的可靠结论。

面向对象量度	描 述
每个类的加权后的方法	这是每个类中的方法数，是一个对每个方法根据复杂度进行加权后所计算得到的。因此，一个简单方法的复杂度为 1，一个大而复杂方法的值将大得多。这个度量值越大，对象类就越复杂。复杂的对象似乎更难以被理解。他们在逻辑上未必是耦合的，因而在一棵继承树中不能有效地作为超类进行复用
继承树的深度（DIT）	这代表在继承树中的具体层数。子类继承超类的属性和操作（方法）。继承树越深，设计就越复杂。很多对象类必须理解后，才能弄清楚树中叶子节点的对象类含义
孩子数（NOC）	这是度量类的直接子类数。它度量类层次结构的宽度，而 DIT 代表它的深度。NOC 高意味着更多的复用。它可能意味着更多的工作量需要用于验证基类，因为依赖于它的子类数大
对象类间耦合度（CBO）	当一个类中的方法使用在另一个类中定义的方法或实例时，类间就是耦合的。CBO 意味着类是高度依赖的，因此在一个类的改变会影响程序中的其他类
对类的响应（RFC）	RFC 是当类的对象接收到消息时潜在可能的对此做出响应的方法数的度量。RFC 是与复杂度相关的。RFC 越高，说明类的复杂度越高，因而它就更容易产生错误
方法中缺乏内聚力（LCOM）	LCOM 是通过计算类中各对方法而得的。LCOM 是两个数的差，一个数是方法间没有共享属性的方法对数，另一个数是方法间共享属性的方法对数。该度量值受到广泛争议，有很多变种。不清楚是否它真的在其他度量所提供的信息上提供了更多的信息

图 24-12 CK 面向对象的度量套件

24.4.2 软件组件分析

作为质量控制过程的一部分，软件度量过程如图 24-13 所示。对系统的每一个组件都使用一系列量度单独分析，对不同组件比较得出的不同量度值，有时还要与以前的项目中收集的历史度量数据进行比较。对异常的度量（严重偏离正常值），可能表示这些组件的质量存在问题。

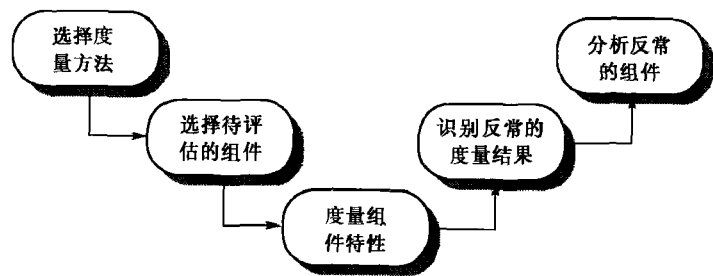


图 24-13 产品度量过程

这个组件度量过程中的几个关键阶段是：

1. **选择要做的度量** 度量要回答的问题应该准确阐述，所需的度量需要良好定义。不必收集与这些问题不直接相关的度量。第 26 章要讨论的 Basin 的 GQM（目标 - 问题 - 量度）范型（Basin 和 Romhach，1988），是在决定要收集什么数据时的一种好方法。
2. **选择要评估的组件** 在软件系统中评估所有组件的量度值既没有必要，也没有意义。在有些情况下，要选择有代表性的组件进行度量。而在其他情况下，则要评估一些特别关键的组件，例如几乎被连续使用的核心组件。
3. **度量组件特性** 度量选出的组件并计算相应的量度值。这一过程中通常使用一个自动化

的数据收集工具对组件的表现形式（设计、代码等）进行处理。这种工具可以是专门写的，也可能是机构所使用的 CASE 工具的某个特征。

4. 识别异常度量 组件度量一旦完成，就应该把它们彼此进行比较，还要把它们与已经记录到度量数据库中的以前的度量相比较。找出每一度量中异常高或者异常低的值，从中可以推测表现出这些数值的组件可能存在问题。

5. 分析异常组件 从特定的量度中一旦识别出具有异常值的组件，就应该检查这些组件，从而确定这些异常量度值是否意味着该组件的质量出现了问题。复杂性的异常量度值并不必然意味着组件的质量差。特别高的数值可能另有原因，它可能不意味着组件的质量出了问题。

应该将收集的数据作为机构的资源保留，即使在特定的项目中并没有使用这些数据，也应保留所有项目的历史记录。一旦一个非常大的度量数据库建立起来，就可以进行跨项目的软件质量比较，并确认内部组件属性与质量特性的关系。

24.4.3 度量歧义

在收集有关软件和软件过程的量化数据时，你必须分析这些数据以了解它们的真正含义。曲解了这些数据就很容易得出错误的结论。不能仅仅只了解数据的本身，还必须同时考虑收集数据的上下文环境。

为了说明对收集到的数据可能存在的不同方式的解释，考虑下面的情景，这个情景和某系统的用户提交的变更请求数有关。

管理者决定监控客户提交的变更请求数。基于这样一种假定，即客户提交的变更请求与产品的可用性和适应性有一定关系，变更请求数越大，软件就越不能满足客户的需要。

处理变更请求和变更软件的费用很高，因此机构决定更改软件过程以提高客户的满意度，同时降低变更的成本。他们希望过程变更会使产品更好，使变更请求减少。

启动过程变更，让软件设计过程中有更多的客户参与。引入对所有产品 Beta 测试，把客户请求的修改反映到交付的产品中去。由这个修改了的过程所生产的新版本就交付给客户了。在有些情况下，变更请求数减少；而在有些情况下，反而增加。这使管理者很困惑，不能评估过程变更对产品质量的影响。

为了了解为什么会发生这种事情，必须了解为什么会提出变更请求：

1. 软件不能做客户想让它做的事情，因此客户通过请求变更来传达他们所要求的功能。
2. 软件非常好，可以被普遍而又频繁地使用。有许多软件客户创造性地想出软件可以完成一些新的功能，因而也可能产生变更请求。

因此，如果客户更多地参与软件开发过程也许会减少客户不满意地方，从而减少变更请求。因为过程变更比较有效，软件的可用性和适应性更好。然而，过程变更可能会不起作用，客户可能已经决定寻找另一个可供选择的系统。产品因出现了竞争对手而失去了市场占有率，也会使得变更请求数减少。这样该产品的用户也就减少了。

另一方面，过程变更可能使许多新的客户更愿意参与产品的开发过程。因此他们会有更多的变更要求。处理变更请求的过程变更可能会加速这种变更的增长。如果公司对客户更负责，他们应该产生更多的变更要求，因为他们知道应该认真对待这些要求。这些公司相信他们的建议极有可能会加入到之后的软件版本中。或者，因为 Beta 测试地点选择不当，没有典型地反映出程序的最大多数使用情况而使变更请求数增加。

要分析变更请求数据，我们不能只知道变更请求的数量。我们需要知道是谁提出变更请求，他们如何使用该软件，以及为什么会提出该请求。我们还要知道一些外部因素，如更改变更请求的程序、市场变化等，是否会对变更请求产生影响。有了上述信息，就有可能揭示出过程变更是

否对提高产品质量有意义了。

以上的论述说明,理解变更影响是困难的,解决这个问题的科学方法是减少那些会影响度量的因素。然而,要度量的过程和产品不能孤立于它们的环境而存在,商业环境是不断变化的,而环境的变化可能使数据的对照失去意义。有关人类活动的量化的数据不能总是看它的表面值。度量值改变的原因往往是模糊的。应该把度量值之所以能够说明产品质量属性的深层次原因调查清楚。

要点

- 软件质量管理就是确保软件有较少的缺陷数,并达到可维护性、可靠性、可移植性等既定标准。质量管理活动包括为过程和产品制定标准,并为检测是否符合这些标准而建立过程。
- 软件标准对质量保证来说非常重要,因为这些标准是对“成功实践”的认同。开发软件时,标准为开发一个优秀质量的软件提供了坚实的基础。
- 机构的质量手册应该编制一套质量保证规程,可以根据 ISO 9001 标准中的通用模型进行编制。
- 对软件过程产生的可交付物进行复查需要有检查质量标准执行情况的团队人员参加。复查是质量评估的一种最为广泛采用的方法。
- 在程序审查或同行评审中,一个小团队系统地检查代码。他们仔细地读这些代码并寻找可能出错和易被遗漏的地方。然后在代码审查会议中讨论这些问题。
- 软件度量可以用于收集有关软件和软件过程的量化数据。所收集的软件量度值可以用于推论产品和过程的质量。
- 产品质量量度对于暴露存在质量问题的异常组件具有特别重要的意义。应该对这些组件做更深入的分析。

进一步阅读材料

《Metrics and Models for Software Quality Engineering, 2nd ed》这是对包括过程和产品,以及面向对象量度的软件度量的非常全面的讨论。也包括一些基于软件评估的需要用以建立和理解模型的数学背景知识 (S. H. Kan, 2003, Addison-Wesley)。

《Software Quality Assurance: From Theory to Implimentation》这本书是一本优秀的、关注于软件质量保证理论与实践。包括对如 ISO 9001 之类标准的讨论 (D. Galin, 2004, Addison-Wesley)。

《A Practical Approach for Quality-Driven Inspections》许多关于审查的文章都相当陈旧,并未考虑现代软件开发实践。这篇相对较新的文章描述了一个审查方法,说明了一些使用审查的问题以及如何在现代开发环境下使用审查的建议。(C. Denger. F. Shull, IEEE Software, 24 (2), March/April 2007)。http://doi.org/10.1109/MS.2007.31。

《Misleading Metrics and Unsound Analysis》一篇优秀的论文,引导度量研究者讨论理解度量的真正意义及面临的困难 (B. Kitchenham, R. Jeffrey and C. Connaughton, IEEE Software, 24 (2), March-April 2007)。http://dx.doi.org/10.1109/MS.2007.49。

《The Case for Quantitative Project Management》这是一本杂志中特别章节的绪论,这本杂志包括其他两篇关于量化项目管理的文章。提出了进一步研究度量和改善软件项目管理的度量的理由 (B. Curtis et al., IEEE Software, 25 (3), May-June 2008)。http://dx.doi.org/10.1109/MS.2008.80)。

练习

- 24.1 解释为什么高质量的软件过程会产生高质量的软件产品。讨论这种质量管理体系可能存在的问题。
- 24.2 解释机构如何用标准去获取关于软件开发的有效率的方法。提出4种可在机构标准中获取的知识。
- 24.3 根据图 27-7 给出的质量属性讨论软件质量评估，依次说明每个属性该如何评估。
- 27.4 设计一个电子表格，使之可以记录复查意见，可以把这些意见以电子邮件的形式发送给复查员。
- 24.5 简要描述可能用到的标准：
 - 在 C、C++ 或 Java 中使用控制结构；
 - 提交一所大学的学期安排项目的报告；
 - 程序变更的构建和批准过程（见第 26 章）；
 - 购买并安装一个新计算机的过程。
- 24.6 假设你所在的工作机构要为微型计算机系统开发数据库产品，而该机构对这个软件开发的量化感兴趣，请你写一份报告提出合适的量度，并说明如何收集量度。
- 24.7 解释为什么程序审查是发现程序错误的一个非常有效的方法？在审查中不可能发现什么类型的错？
- 24.8 为什么设计量度就其自身而言不是预测设计质量的好方法？
- 24.9 解释为什么确认内部产品属性（如环路复杂度）与外部属性（如可维护性）之间的关系是困难的。
- 24.10 有个同事是很棒的程序员，做的软件错误数很少，但她一贯忽视机构的质量标准。机构的管理者该如何对待这种行为？

参考书目

- Bamford, R. and Deibler, W. J. (eds.) (2003). 'ISO 9001:2000 for Software and Systems Providers: An Engineering Approach'. Boca Raton, Fla.: CRC Press.
- Barnard, J. and Price, A. (1994). 'Managing Code Inspection Information'. *IEEE Software*, **11** (2), 59-69.
- Basili, V. R. and Rombach, H. D. (1988). 'The TAME project: Towards Improvement-Oriented Software Environments'. *IEEE Trans. on Software Eng.*, **14** (6), 758-773.
- Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., Macleod, G. and Meritt, M. (1978). *Characteristics of Software Quality*. Amsterdam: North-Holland.
- Chidamber, S. and Kemerer, C. (1994). 'A Metrics Suite for Object-Oriented Design'. *IEEE Trans. on Software Eng.*, **20** (6), 476-93.
- Crosby, P. (1979). *Quality is Free*. New York: McGraw-Hill.
- El-Amam, K. 2001. 'Object-oriented Metrics: A Review of Theory and Practice'. National Research Council of Canada. <http://seg.iit.nrc.ca/English/abstracts/NRC44190.html>.
- Endres, A. and Rombach, D. (2003). *Empirical Software Engineering: A Handbook of Observations, Laws and Theories*. Harlow, UK: Addison-Wesley.
- Fagan, M. E. (1976). 'Design and code inspections to reduce errors in program development'. *IBM Systems J.*, **15** (3), 182-211.
- Fagan, M. E. (1986). 'Advances in Software Inspections'. *IEEE Trans. on Software Eng.*, **SE-12** (7), 744-51.
- Gilb, T. and Graham, D. (1993). *Software Inspection*. Wokingham: Addison-Wesley.
- Grady, R. B. (1993). 'Practical Results from Measuring Software Quality'. *Comm. ACM*, **36** (11), 62-8.

Gunning, R. (1962). *Techniques of Clear Writing*. New York: McGraw-Hill.

Hall, T. and Fenton, N. (1997). 'Implementing Effective Software Metrics Programs'. *IEEE Software*, **14** (2), 55–64.

Humphrey, W. (1989). *Managing the Software Process*. Reading, Mass.: Addison-Wesley.

IEC. 1998. 'Standard IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems'. International Electrotechnical Commission: Geneva.

IEEE. (2003). *IEEE Software Engineering Standards Collection on CD-ROM*. Los Alamitos, Ca.: IEEE Computer Society Press.

Ince, D. (1994). *ISO 9001 and Software Quality Assurance*. London: McGraw-Hill.

Kilpi, T. (2001). 'Implementing a Software Metrics Program at Nokia'. *IEEE Software*, **18** (6), 72–7.

Kitchenham, B. (1990). 'Measuring Software Development'. In *Software Reliability Handbook*. Rook, P. (ed.). Amsterdam: Elsevier, 303–31.

McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction, 2nd edition*. Seattle: Microsoft Press.

Mills, H. D., Dyer, M. and Linger, R. (1987). 'Cleanroom Software Engineering'. *IEEE Software*, **4** (5), 19–25.

Offen, R. J. and Jeffrey, R. (1997). 'Establishing Software Measurement Programs'. *IEEE Software*, **14** (2), 45–54.

Stalhane, T. and Hanssen, G. K. (2008). 'The application of ISO 9001 to agile software development'. *9th International Conference on Product Focused Software Process Improvement, PROFES 2008*, Monte Porzio Catone, Italy: Springer.

配置管理

目标

本章的目标是介绍软件配置管理过程和工具。读完本章，你将了解以下内容：

- 了解软件变更管理的过程和规程；
- 了解版本管理系统必须提供的重要功能以及版本管理和系统构建之间的关系；
- 了解系统版本和系统发布的区别，了解发布管理过程的各个阶段。

在开发和使用过程中软件系统常常会变更。必须发现错误并加以修正。系统需求变更后，开发者不得不在新版本中实现这些变更。有了新版本的硬件和系统平台之后，开发者不得不使自己的系统与之兼容。竞争者在他们的系统中引入新的特性时，你一定也要做出相应的调整。当软件发生变更时，一个新的版本就产生了。因此，大多数系统都有一系列的版本，每一个版本都需要进行维护和管理。

配置管理（CM）与管理变更软件系统的政策、过程和工具有关。进化中的系统之所以需要管理，是因为系统在进化时会产生许多不同的版本，这些版本包含了变更提议、错误修正以及对不同硬件和操作系统的适应等诸多内容。可能有几个版本同时开发、同期使用。这样就需要跟踪已经实现的变更以及这些变更是怎样包含在软件产品中的。如果没有有效的配置管理规程，就可能浪费精力修改一个错误的系统版本或发布一个错误版本给用户，甚至不知道特定系统或组件源代码存放在什么地方。

配置管理对个人项目管理来说非常有用，因为开发者容易忘记已做过的变更。对于几个开发者同时完成一个软件系统的团队项目配置管理也是必要的。有时这些开发者全都工作在一个地方，但是，越来越多开发团队的成员分散在世界的各个角落。配置管理系统的使用确保了团队能够获得正在开发系统的信息而不妨碍彼此的工作。

一个软件系统产品的配置管理包括4个紧密相关的活动（见图25-1）：

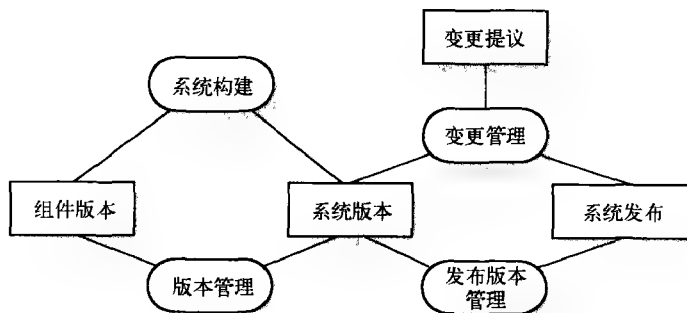


图 25-1 配置管理活动

1. **变更管理** 包括跟踪来自客户和开发者的软件变更请求，计算做出这些变更的花费并估计其影响，决定是否变更、何时完成变更。

2. **版本管理** 包括跟踪系统组件的多个版本，确保由不同开发者对组件做出的变更不会彼此干涉。

3. **系统构建** 是一个组装程序组件、数据和库的过程，然后把这些组件编译链接成一个可执行系统。

4. **发布管理** 包括准备对外发布的软件，持续跟踪已经发布以供客户使用的系统版本。

配置管理包括处理海量信息和许多为了支持配置管理而开发的配置管理工具。这些工具包括简单工具，用来支持单一的配置管理任务，例如错误追踪；也包括复杂昂贵的集成工具，用来支持所有的配置管理活动。

配置管理政策和过程规定了如何记录和处理所提议的系统变更，如何决定需要变更的系统组件，如何管理系统的不同版本及其组件，如何向客户发布变更。配置管理工具用来追踪变更提议，存储系统组件的多个版本，从这些组件中构建系统，并跟踪系统版本的实际发布。

有时配置管理被视为广义的软件质量管理过程的一部分（软件质量管理过程在第 24 章已经讨论过）。质量管理和配置管理可能是由同一个管理人员负责的。当一个软件的新版本完成后，开发人员把软件交给质量保证团队，由他们负责检查系统的质量是否合乎要求。如果符合要求，则该系统变为一个受控的系统，也就意味着在系统所有变更实现之前，它们必须经过一致认可和记录。

配置管理标准的定义和使用对 ISO 9000、CMM 及 CMMI 标准（Ahern 等，2001；Bamford 和 Deibler，2003；Paulk 等，1995；Peach，1996）的质量认证是至关重要的。这些 CM 标准基于由一些主体（例如 IEEE）开发的通用 CM 标准。IEEE 828 - 1998 就是一个配置管理计划的标准。这些标准集中于 CM 过程和 CM 过程中产生的文件。在外部标准的基础上经过加工剪裁，可以形成更加具体的适于专门环境的机构标准。

配置管理的难题之一是不同的公司使用不同的术语称呼相同的概念。这是有历史原因的。军用软件系统可能是第一个使用配置管理的系统，这些系统术语反映了已经存在的硬件配置管理的过程和流程。商业系统的开发者对军事过程和术语不熟悉因此发明了适合自己的术语。敏捷方法也设计了新的术语，有时引入专门的术语来区别敏捷方法和传统的 CM 方法。图 25-2 解释了本章使用的配置管理术语。

术 语	解 释
配置项或软件配置项（SCI）	与配管理控制下的软件项目有关的事物（设计、代码、测试数据、文档等）。配置项会存在多个不同的版本。每个配置项都有一个唯一的名字
配置控制	确保系统和组件的版本得到记录和维护的过程。这样变更就会得到管理，所有的组件的版本都能在整个系统生命期中识别和存储
版本	配置项的一个实例，区别于其他配置项的实例。版本总是有一个唯一的标识符，通常由配置项名字加上版本号组成
基线	基线是用于组成系统的组件版本的集合。基线是受控的，意味着构成系统的组件的版本是不能改变的。我们总是可以从它的组成组件中重新创建一个基线
代码线	代码线是软件组件以及组件所依赖的其他配置项的集合
主线	代表系统不同版本的基线的序列
发布版本	发布给客户（或其他机构中用户）使用的系统版本
工作空间	一个私有的工作空间，在其中软件可以修改而不至于影响其他会使用或修改软件的开发者
分支	从现存的代码线的版本中创建一个新的代码线。然后新的代码线和已经存在的代码线可以独立开发
合并	通过合并在不同代码线中的单独版本创建软件组件的新版本。这些代码线可能是由某个代码线的先前存在的分支所创建的
系统构建	通过耦合和链接组件和库的适当版本创建一个可执行的系统版本

图 25-2 配置管理术语

25.1 变更管理

变更对大型软件系统而言是一种无法更改的事实。在系统的整个生命周期内机构的需要和需求发生改变，错误需要修正，系统需要适应变化了的环境。为了确保变更以一种可控制的方式应用在系统中，开发者需要一系列工具的支持和变更管理程序。变更管理确保系统的进展是一个可管理的过程，并且最紧急和费效最优的变更拥有最高的优先权。

变更管理过程主要关心的是对所提议的变更的成本收益分析，保证变更是值得去做的，并且记录系统的哪些组件已经改变。图25-3 是一个变更管理过程的例子，它显示了主要的变更管理活动。有许多不同的此种过程的变种在使用，但是，为了有效起见，变更管理过程总是包含检查方法、成本估计和变更确认等内容。当软件交付给客户用来发布或是提交给一个机构用来部署时，变更管理过程就开始启动了。

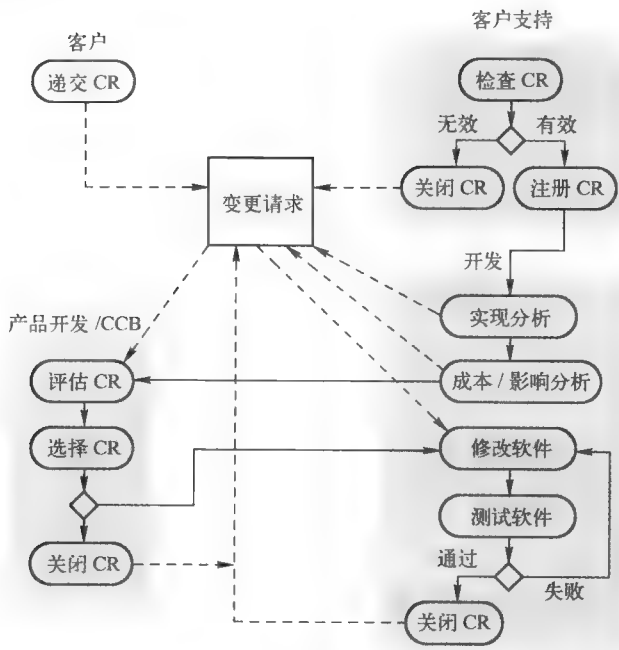


图 25-3 变更管理过程

当一名“客户”完成并提交一个描述对系统的变更的变更请求时，变更管理过程就开始了。这可能是一个描述错误症状的错误报告，也可能是增加系统附加功能的请求。一些公司分开处理错误报告和新的请求，但原则上它们都是简单的变更请求。变更请求可以通过填写变更请求表（CRF）来提交。在这里使用“客户”来概括其他不是开发团队的人，因此变更可以由公司其他部门提出，例如市场营销部门。

电子变更请求表记录了可以由所有变更管理小组分享的信息。当处理变更请求时，添加到变更请求表中的信息记录了处理过程的每一阶段所做的决定。因此，在任何阶段 CRF 代表变更请求状态的一个快照。除了记录需要的变更之外，CRF 还要记录有关变更的提议、变更的估算成本，以及变更的请求、核准、实现和有效性验证等日期。另外 CRF 还可能包括的一个部分就是开发人员对如何实现变更的概述。

图 25-4 是一个变更请求表的例子，图中只给出了一部分内容。这是在一个大型复杂系统设

计项目中使用的 CRF 的一个例子。对于小的项目，建议的变更请求要得到正式记录，但是 CRF 可以集中在对所要求的变更的描述上，而无需对实现问题给以太多的描述。作为一名系统开发者，必须决定如何实现变更以及估计实现变更所需的时间。

变更请求表	
项目: SICSA/AppProcessing	Number: 23/02
变更请求者: I. Sommerville	Date: 2009-01-20
请求的变更: 申请者的状态应该显示在申请人列表中	
变更分析人: R. Look	分析日期: 25/01/09
受影响的组件: ApplicantListDisplay, StatusUpdater	
相关的组件: StudentDatabase	
变更评估: 实现比较简单, 可以根据状态改变显示颜色。需要增加一个表将状态和颜色关联起来, 无需修改任何相关组件	
变更优先级: 中	
变更实现:	
估计的工作量: 2 小时	
到 SGA 应用团队的日期: 28/01/09	CCB 决定日期: 30/01/09
决定: 接受变更。变更将在版本 2.1 中实现。	
变更实现者:	变更日期:
向 QA 提交的日期:	QA 决定:
向 CM 提交的日期:	
注释	

图 25-4 部分完成的变更请求表



客户和变更

敏捷方法强调在变更优先权选择过程中客户参与的重要性。客户代表帮助团队决定在下一个开发循环中需要实现的变更。尽管这对于像那些为单个客户开发的系统来说是有效的，它对于那些没有真正的客户与团队一起工作的产品是有问题的。在这样的情况下，团队需要自己决定变更的优先权顺序。

<http://www.SoftwareEngineering-9.com/Web/CM/agilechanges.html>

变更请求提交后，首先需要进行检查以确保是否有效。检查者可以来自于客户、应用支持团队或者是开发团队的一个成员（内部请求时）。因为并不需要响应所有的变更请求，所以检查是必要的。如果变更请求是一个错误报告，这个错误可能已经记录过了。有时，用户认为的问题实际上是由于对系统的误解而提出的。有时，用户请求一些已经完成的属性，但他们并不知道有关这些属性的知识。如果上述任一种情况出现，那么变更请求就结束了，并且以结束的原因来更新表格。如果变更请求是有效的，那么就为随后的分析记录一个显著的请求。

对于有效的变更请求，变更管理过程的下一个阶段就是对变更进行评估和成本估算。这通常是开发团队或者维护团队的任务，因为他们能够计算出完成变更所需的成本。变更对系统其他部分带来的影响必须逐一核查，因此开发者必须检查所有因变更而受到影响的组件。如果变更意味着需要对系统其他部分的进一步变更，那么这明显会增加完成变更的成本。接下来评估系统请求变更的模块。最后估算实现变更的成本以及其他系统组件可能要相应发生变更的成本。

由以上分析可知，应该由一个专门的小组来决定做出软件变更是否是划算的。对于军用和政府系统，这个小组常被称为变更控制委员会（CCB）。工业上可能被称为“产品开发小组”，

负责决定一个软件系统如何进展。除非变更只包括改正屏幕显示或文件中的小错误,否则都应该提交给这个小组,由它来决定是否应该同意变更。当分析比完成变更花费更多时,应该无须通过分析就通过开发团队的变更请求。

变更控制委员或是产品开发小组应从战略的、机构的角度,而不应从技术角度去考查变更带来的影响。由它来决定该变更在经济上是否合理,机构内部是否有同意变更的充分理由。已同意的变更请求反馈给开发团队,驳回的变更请求就此结束而无需采取进一步行动。在决定是否同意变更请求时需要考虑的重要因素有:

1. **不做变更会引起的后果** 当评估一个变更请求时,必须考虑如果变更没有完成将会发生什么。如果变更和系统错误有关,则变更失败的严重性必须要考虑到。如果系统错误引起系统崩溃,这是非常严重的,可能会影响系统的正常使用。另一方面,如果错误的后果影响较小,比如显示颜色的错误,那么并不需要立即修正错误,因此变更只需一个较低的优先权。

2. **变更的益处** 变更是否使系统的许多用户受益?或者仅仅只令变更的提议者受益?

3. **变更影响的用户数** 只要有一部分用户受到影响,就给变更分配一个较低的优先权。事实上,当变更可能对大多数的系统用户有不利影响时,则该变更是不可取的。

4. **变更所需花费** 如果变更影响许多系统组件(因此增加了引入新错误的机会),并且/或者完成变更需要花费大量的时间,那么评估变更花费后该变更可能被驳回。

5. **产品发布循环** 如果软件的一个新版本刚刚发布完,那么推迟变更直到计划发布下一个版本时的做法是有意义的。

对软件产品的变更管理(例如CAD系统产品)比为特定客户开发的软件系统的变更管理的处理上稍有不同。对软件产品,客户并不直接参与进来,所以变更与客户业务的相关性不是问题。这些产品中的变更请求来自客户支持团队、公司市场营销团队和开发团队自身。这些需求可能反映来自客户的建议和反馈,或者是由竞争产品提供的分析。

客户支持团队提交的变更请求可能与软件发布后由客户发现并报告的错误相联系。客户可能通过网页或是电子邮件来报告缺陷。缺陷管理小组确认缺陷并把它们写成正式的系统变更请求。市场工作人员会见客户并调查竞争产品,他们建议的变更可能会包括如何更容易地向新客户推销系统的新版本。系统开发者可能会有一些可以添加到系统中的新属性的好想法。

图25-3显示了系统向客户发布后的变更请求过程。开发期间,当系统的新版本通过每日的系统构建创建以后,就可以采用较为简单的变更管理过程了。出现的问题和进行的变更仍然要记录下来,但是只影响到单个组件和模块的变更,不需要单独评估,直接把它们送到系统开发人员手中即可。他们或者接受它们,或者论证为什么这些变更是不必要的。当变更影响到由不同开发团队生产的系统模块时,还是应该由某些变更控制权威人士(例如系统构建者)进行评估,由他们决定这些变更的优先权。

在一些敏捷方法中,例如极限编程中,客户直接参与决定是否实现变更。当他们对系统需求提出变更时,他们与项目组成员一起评估变更产生的影响,然后决定这个变更是否应该优先于为系统下一个增量所规划的特征。然而,涉及系统改善的变更就留给系统编程人员来决定了。对已经完成的代码进行改进的过程,是软件不断得到改善的过程,它不能看成是一项经常开支,而是开发的一个必要部分。

当开发团队变更软件组件时,他们应该维护每个组件的变更记录,有时把这称为组件的导出历史。保持导出历史的最佳方式是将其放在组件源代码开头的标准化的注释部分(见图25-5)。这些注释应该索引到引起系统变更的请求。于是你就可以写出扫描所有组件的简单脚本并处理导出历史来产生组件变更报告。对于文档,在每个版本中的变更记录经常放在一个独立页中,一般放在文档的前面部分。在关于文件的网页章节讨论了这些主题。

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: R. Looek
// Creation date: 13/11/2009
//
// © St Andrews University 2009
//
// Modification history
// Version  Modifier  Date      Change      Reason
// 1.0      J. Jones  11/11/2009  Add header  Submitted to CM
// 1.1      R. Looek  13/11/2009  New field   Change req. R07/02
```

图 25-5 导出历史

变更管理由专门的软件工具支持。这些可能是相对简单的基于网页的工具，比如说 Bugzilla。它用许多开放的资源系统来报告问题。相对而言，复杂工具的使用令变更处理自动化，包括从最初的客户提议到批准变更的整个过程。

25.2 版本管理

版本管理（Version Management，VM）是跟踪软件组件或配置信息以及使用这些组件系统的不同版本的过程。版本管理也包括确保由不同开发者做出的变更不会彼此影响。因此，可以把版本管理过程看做是管理代码线和基线的过程。

图 25-6 说明了代码线和基线之间的差别。本质上，代码线就是源代码版本的序列，一个晚期的版本是由某个早期版本发展而来。代码线通常应用于组件以便每个组件有不同的版本。基线是对一个特殊系统的定义。因此基线指的是包含在系统中的组件版本加上所使用的库的描述、配置文件等。由图 25-6 可以看出，不同的基线使用来自各个代码线的不同组件版本。图中的阴影方盒代表了定义在基线中的组件，表明实际上引用了代码线中的组件。主线是由一个原始基线发展而来的系统版本序列。

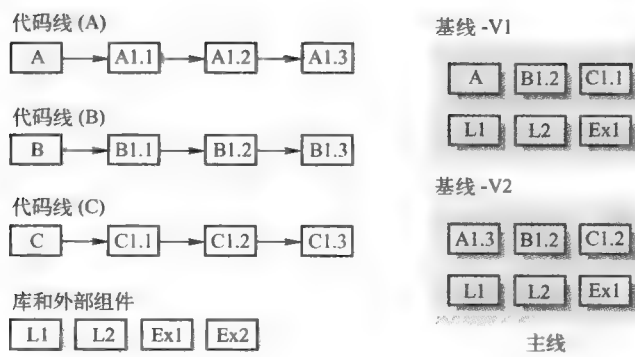


图 25-6 代码线和基线

可能会使用一种配置语言来详述基线，以使用户定义一个特殊系统版本所包括的组件。精确地描述一个组件版本或者仅仅只是描述组件标识符都是可行的。若使用标识符，则意味着在基线中应该使用组件的最近版本。

基线很重要，因为开发者常常不得不重建一个完整系统的特定版本。例如，一个产品线需要实例化为不同客户产生不同的个人系统版本。假如客户报告了系统中错误，那么开发者不得不重建这个版本。

为了支持版本管理，应该经常使用版本管理工具（有时称为版本控制系统或者源代码控制系统）。这些工具识别、存储并控制不同版本的存储。有许多可使用的版本管理系统，包括被广泛使用的开源系统，比如 CVS 和 Subversion。

版本管理系统通常提供一系列特征：

1. **版本和发布版本识别** 被管理版本提交给系统时给它们分配标识符。这些标识符通常基于配置项的名字（例如 ButtonManager），后跟一位或几位数字。因此 ButtonManager 1.3 表示此组件代码线 1 的第三个版本。一些 CM 系统也允许关联上版本的属性（例如 mobile, smallscreen），这些也能够用来鉴别版本。一个一致的识别系统是重要的，因为它简化了定义配置的问题。它也使快速索引更简单（例如，*.V2 意为所有组件的版本 2）。

2. **存储管理** 为了减少只有轻微差异的不同版本所占存储空间，版本管理系统通常会提供存储管理工具。系统只存储每个版本不同之处的列表而不是每个版本的副本。通过把这些应用到源版本（通常是最近的版本）上，就能够重建目标版本。

3. **变更历史记录** 记录并列出现所有对系统或组件做出的变更。在一些系统中，这些变更可能用来选择一个特殊的系统版本。我们可以用描述所做的变更的关键词来标记组件。然后就可以用这些标记来选择包括在基线中的组件。

4. **独立开发** 不同的开发者可能同一时间正在相同的组件上工作。版本管理系统跟踪检出的组件，确保由不同开发者对组件做出的变更不会彼此影响。

5. **项目支持** 一个版本管理系统可能支持共享组件的几个项目的开发。在项目支持系统（比如 CVS（Vesperman, 2003））中，可能检入和检出所有与项目相关的文件，而不是一次只能在一个文件或目录上工作。

当首次开发版本管理系统时，存储管理是其最重要的功能之一。版本控制系统中的存储管理特征减少了维护所有系统版本所需的硬盘空间。当创建一个新版本时，系统只创建新版本和较旧版本之间的不同之处的一个列表，然后根据这个列表来创建新版本（如图 25-7 所示）。在图 25-7 中，阴影的部分代表早期组件版本，它由最近组件版本自动创建。增量备份用来存储变更代码行，并应用这些代码行自动由一个版本创建另一个版本。最近的组件版本最可能被调用，因此大多数系统完整地存储最近的组件版本。增量备份定义怎么重建早期的系统版本。

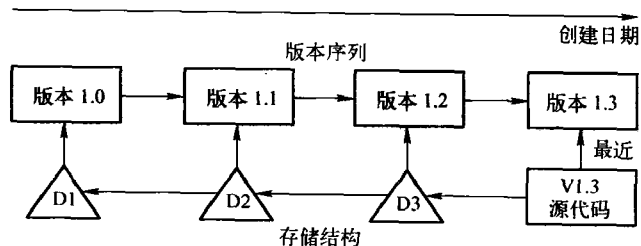


图 25-7 使用增量的存储管理

大多数的软件开发是一个团队活动，因此经常会有不同的团队在同一时间开发同一组件的情况。例如，假设 Alice 对系统做出一些变更，包括变更组件 A、B 和 C。与此同时，Bob 需要变更组件 X、Y 和 C。因此 Alice 和 Bob 都需要变更组件 C。这些变更不会影响到彼此是非常重要的——Bob 的变更覆盖 Alice 的变更，反之亦然。

为了使彼此之间不相互干涉，支持独立开发，版本管理系统使用一个公共仓库和一个私有工作空间的概念。开发者从公共仓库把组件取到他们的私有工作空间，并且可以在私有工作空间中按照他们的意愿来变更组件。当变更完成后，把组件再放回公共仓库中。图 25-8 说明了这一点。如果两个或更多的开发者同一时间开发同一组件，每个开发者必须检查仓库中的组件。

如果取出一个组件，版本管理系统将会警告其他使用者该组件已经被取出。当放回的组件被修改时，系统将会确保给不同的版本分配不同的标识符并把他们分开存放。

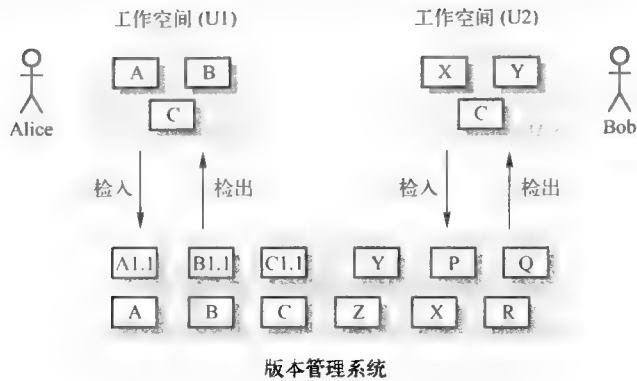


图 25-8 从版本容器中检入和检出

相同组件独立开发的一个后果是代码可能会出现代码线分支。可能会有几个独立的版本序列，而不是反应变更的一个线性版本序列，如图 25-9 所示。不同的开发者独立地在不同的源代码版本上工作并以不同的方式做出改变，这在系统开发中是很常见的。

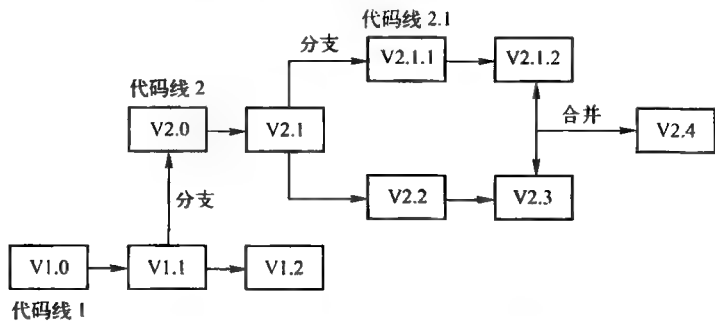


图 25-9 分支和合并

在某一阶段，合并代码线分支以创建一个组件的新版本，使其包括所有已做的变更，这可能是必要的。这也在图 25-9 示意出。组件版本 2.1.2 和 2.3 合并创建了组件 2.4。如果变更发生在代码的完全不同的部分，版本管理系统会通过对应应用到代码的增量做混合使组件版本自动合并。更常见的情况是，所做变更会有重叠部分，它们会影响到彼此。开发者必须检查出冲突并且修改变更以使它们相兼容。

25.3 系统构建

系统构建是把软件组件、外部的库、配置文件等编译和链接成一个完整的、可执行的程序的过程。系统构建工具和版本管理工具必须进行通信，因为构建过程包括在版本管理系统管理的知识库中检查组件版本信息。用于识别基线的配置描述信息也用于系统构建工具。

构建是一个复杂的过程，很容易出现错误。可能有 3 种不同的系统平台（见图 25-10）：

1. 开发系统，包括开发工具，比如编译器、源码编辑器等。开发人员将代码从版本管理系统下载到私有工作空间，再做修改。他们希望在自己的开发环境中建立一个系统版本以便测试，然后再提交做出的修改到版本管理系统。这需要在私有工作空间使用那些本地构建工具去处理检出的组件版本。

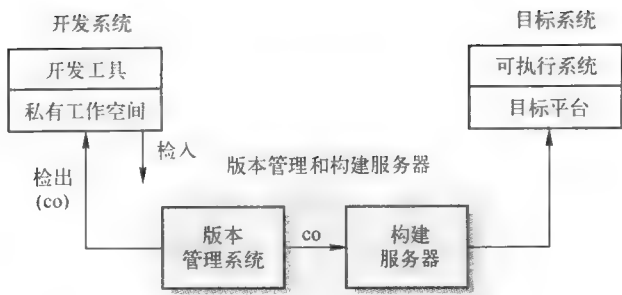


图 25-10 开发、构建和目标平台

2. 构建服务器，用于构建确定的、可执行的系统版本。它和版本管理系统紧密交互。开发人员在系统版本被构建之前将代码检入给版本管理系统。系统构建可能依赖外部库，而这些库并未包含在版本管理系统中。

3. 目标环境，这是系统运行的平台。这可能是和用于开发和构建系统相同类型的电脑。然而，对于实时和嵌入式系统来说，目标环境相对于开发环境常常更微小更简单（比如手机）。对于大型系统，目标环境可能包含数据库以及其他 COTS 系统，而这不能安装在开发机器上。在这两种情况下，在开发电脑或者构建服务器上构建和测试系统都是不可能的。

开发系统和构建服务器都可以和版本管理系统进行交互。可以将虚拟机系统安装在构建服务器上或者一个专门的服务器上。对于嵌入式系统，可能在开发环境中安装一个模拟环境进行测试，而不是使用实际的嵌入式系统平台。这些模拟机器能够提供比嵌入式系统更好的调试支持。然而，在每个方面模拟嵌入式系统的行为都是很困难的。因此，必须在执行系统的实际平台上运行系统测试，同时也在系统仿真器上执行系统测试。

系统构建就是将软件有关的大量信息和它的运行环境装配起来。因此，除了非常小的系统之外，使用自动构建工具进行系统构建是很有意义的（见图 25-11）。注意，在构建中不只是需要源代码文件，可能还必须将其与外部提供的库、数据文件（比如错误消息的文件）以及定义目标安装的配置文件相链接。可能还必须指明构建所要使用的编译器和其他软件工具的版本。理想情况下，构建一个完整的系统只需要一个命令或者轻点一次鼠标。

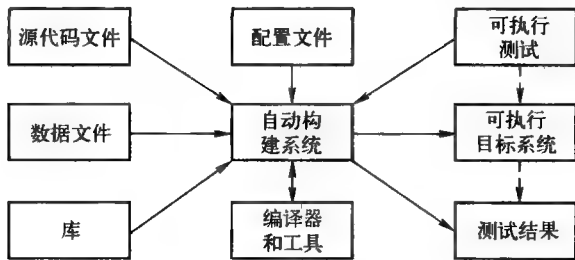


图 25-11 系统构建

有很多可用的构建工具，一个构建系统可能提供部分或者全部以下特征：

1. **构建脚本生成** 在必要的情况下，构建系统应该分析待构建的程序，识别依赖的组件，并且自动生成一个构建脚本（有时候叫做配置文件）。系统也应该支持手工创建和编辑构建脚本。
2. **版本管理系统集成** 构建系统应该从版本管理系统中检出需要的组件版本。
3. **最小化再编译** 构建系统应该分析出哪些源代码需要进行再编译，再对需要的代码进行编译。

4. **可执行系统创建** 构建系统应该将编译后的各个目标代码文件以及其他需要的文件（比如库文件、配置文件）链接起来，创建可执行的系统。

5. **测试自动化** 有的构建系统能够使用自动化工具，如 JUnit，自动运行自动测试。这样能够检查构建是否被变更所破坏。

6. **报告** 构建系统应该提供关于运行的构建或者测试成功与否的报告。

7. **文档生成** 构建系统可能能够生成关于构建和系统帮助页面的版本注释。

构建脚本是一个对要构建的系统的定义。其中包含组件的信息以及它们之间的依赖关系，用于编译和链接系统的工具的版本信息。构建脚本包括配置详细描述，所以所使用的脚本语言通常和配置描述语言一样。配置语言包括描述待包含在构建中的系统组件以及它们之间的关系的结构。

由于编译是一个计算量密集的过程，支持系统构建的工具通常设计使所需要的编译量最小化。为达到这一目的，需要检测是否存在可用的已编译版本。如果存在，就不需要重新编译这个组件。因此，需要有一种方法，无二义地将组件的源代码和相对应的目标代码联系起来。

解决这个问题的方法是：赋予存储源代码组件的文件一个独一无二的签名，从这个源代码编译的目标文件，也赋予一个相关的签名。签名能够识别各个源代码版本，当修改源代码后签名也改变。通过比较源代码和目标代码文件的签名，就能够决定是否使用源代码组件产生目标代码组件。

存在两类可以使用的签名：

1. **修改的时间戳** 源代码文件的签名是文件修改的时间日期。如果组件的源代码文件在相关的目标代码文件之后修改，系统就认为需要重新编译以生成新的目标代码文件。

比如，组件 `Comp.java` 和 `Comp.class` 的修改时间戳分别是 17:03:05:02:14:2009 和 16:34:25:02:12:2009。这表示 Java 代码修改时间是 2009 年 2 月 14 号 17 点 3 分 5 秒，编译版本的修改时间是 2009 年 2 月 12 号 16 点 34 分 25 秒。这种情况下，编译版本没有包括 2 月 12 后之后对源代码的修改，系统会自动重新编译 `Comp.java`。

2. **源代码校验和** 源代码文件的签名是从文件中数据计算出的校验和。校验和函数使用源代码文件作为输入，计算出一个独一无二的值。如果改变源代码（即使是一个字符），就将生成不同的校验和值。这样就肯定能够判断有着不同的校验和的文件是不相同的。在编译之前给源代码赋予校验和唯一的标识源文件。然后构建系统将校验和标签贴在生成的目标代码文件上。如果没有相同标签的目标代码存在，就需要重新编译源代码。

由于目标代码文件没有正规的版本标识，第一种方法意味着只有最新编译的目标代码文件保存在系统中。目标文件一般通常通过名字和源代码文件联系起来（比如，和源代码文件有相同的名字，但是加上不同的后缀）。因此，源文件 `Comp.java` 产生目标文件 `Comp.class`。因为源文件和目标文件通过名称联系起来，而不是明确的源文件签名，由于要生成相同名字的目标文件，一般不可能在同一个目录同一时间建立源代码组件的不同版本。

校验和方法的优点是允许同时保留组件目标代码的多个不同版本。签名（而不是名字）将源代码和目标代码联系起来。源代码和目标代码有相同的签名。因此，重新编译的时候，不会改写原来的目标代码，而时间戳的方式中会覆盖原来的代码。编译会生成新的目标代码文件，并贴上源代码的签名。并行编译是可以的，组件的不同版本可以同时进行编译。

敏捷方法建议：应该使用自动测试（有时称“烟雾测试”）执行那些非常频繁的系统构建，从而发现软件问题。频繁构建可能是连续集成过程（见图 25-12）的一部分。为了配合敏捷方法的、进行小的改变的理念，持续集成包括在对源代码做出小的改变后，频繁重构主线。持续集成的步骤如下：

1. 将主线系统从版本管理系统中检出到开发人员的私有工作空间。
2. 构建系统并运行自动测试以确保所构建的系统能够通过所有测试。如果没能通过，构建终止。这时应该通知最后提交主线系统的开发人员。他们负责修复这个问题。
3. 完成系统组件的变更。
4. 在私有工作空间构建系统并重新运行测试。如果测试失败，继续编辑。
5. 一旦系统通过测试，将它检入到构建系统，但是不要作为新系统基线提交。
6. 在构建服务器上构建系统并运行测试。这样做是因为在检出系统后其他人有可能修改了组件。如果确实如此，检出失败的组件并进行编辑，使测试在私有工作空间通过。
7. 如果系统在构建系统上通过测试，将做出的变更作为系统主线中的一个新的系统基线。

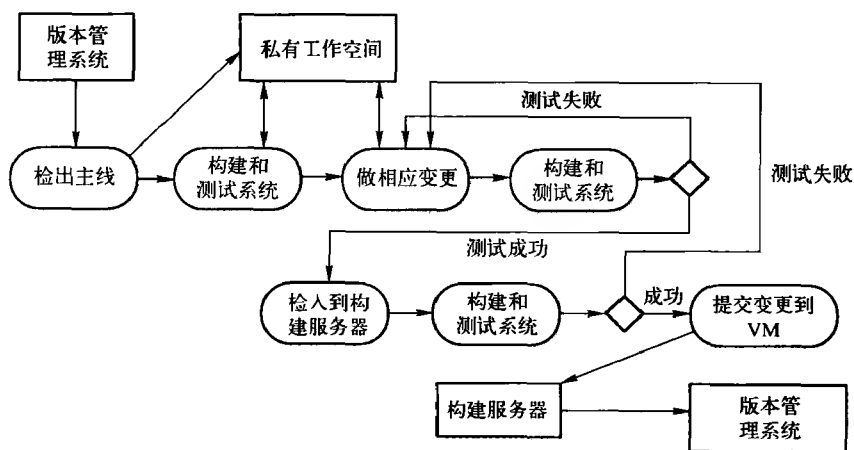


图 25-12 持续集成

持续集成的优点是，它允许尽快发现并且修复由不同开发人员的交互引起的问题。主线中最近的系统是确定的工作系统。然而，虽然持续集成是一个好想法，但在系统构建中实现这个方法并不总是可行的。原因如下：

1. 如果系统很庞大，构建和测试可能要花费大量时间。因此不太可能每天构建系统几次。
2. 如果开发平台和目标平台不相同，就不大可能在开发人员的私有工作空间中运行系统测试。硬件、操作系统或者安装的软件都可能存在差异。因此需要更多的时间测试系统。

对于大型系统或者执行平台和开发平台不相同的系统来说，持续集成可能不太实际。在那些环境下，可以使用每日构建系统。其特征如下：

1. 开发机构设定系统组件交付时间（比如下午 2 点）。如果开发人员有正在编写的组件的新版本，他们必须在下午 2 点前提交。组件可能并不完整但是应该提供一些能够测试的基本功能。
2. 通过编译和链接这些组件构建系统的新版本，形成完整的系统。
3. 将系统交付给测试组，执行一系列预定的系统测试。同时，开发人员仍然继续编写组件，增加新功能，并修复之前测试发现的错误。
4. 记录系统测试中发现的问题，并交还给系统开发人员。他们在随后的版本中修复这些问题。

使用软件频繁构建的好处是，找到一些问题的几率增加了，这些问题源于过程早期组件的交互。频繁构建鼓励组件的彻底的单元测试。心理学上，开发人员都位于不要“破坏构建”的压力之下，即他们都尽量避免检入引起整个系统崩溃的组件版本。因此他们不愿提交未能充分测试的新的组件版本。结果是，花在系统测试发现和处理软件错误上的时间更少。

25.4 发布版本管理

系统的发布版本是分发给客户的系统版本。对于大众市场软件，通常可能定义两种类型的发布，一种是主要发布，用于交付重要的新功能；另一种是小型发布，用于修复漏洞和修复用户报告的问题。比如，这本书在苹果 Mac 电脑上书写的，操作系统是 OS 10.5.8。这表示操作系统 OS 10 的第 5 次主要发布的第 8 次小型发布。在经济上来说，主要发布对于软件卖主非常重要，因为顾客必须对此付款。小型发布通常是免费分发。

对于定制软件或者软件产品线，管理系统发布是很复杂的过程。系统的特别发布版本可能必须为每个客户或者个人客户生产，可能同时运行几种不同的系统版本。这表示销售专门软件产品的软件公司可能必须管理产品的数十种甚至数百种不同的发布版本。他们的配置管理系统和过程必须设计成能提供如下信息：哪位客户使用何种系统发布版本，以及发布版本和系统版本的关系。如果发生问题，可能需要重新生产已经交付给特定客户的软件。

因此，一个系统的发布版本生成时，必须编制文档以保证将来可以重新准确地复制它。这一点对定制的、生命周期长的嵌入式系统尤为重要。客户可能很多年都使用一个这样的系统，在距离最初发布很长时间之后才对某一特定的软件发布版本提出具体的变更需求。

为发布版本编制文档，必须记录用来产生可执行代码的源代码组件的特定版本。也要保存源代码文件、相应可执行代码以及所有的数据和配置文件。还应该记录操作系统的版本、库、编译程序和建立软件的其他工具。这些在日后建立完全相同的系统时用得上。在这种情况下，软件平台和工具的拷贝也要在版本管理系统中保存。

准备并分发一个系统版本是个很高代价的过程，尤其是那些大众市场的软件产品。在准备发布版本的时候，除了必须准备的技术工作之外，还有广告、宣传材料以及到位的市场策略，说服消费者购买新的系统版本。必须认真考虑发布的时间间隔。如果发布过于频繁或者要求硬件升级，客户就可能不愿意升级到新版本，特别是当他们需要付费的时候；如果发布间隔时间较长，会失掉市场份额，因为客户会购买其他的系统。

决定何时发布系统的一个新版本，应该通盘考虑技术和机构的各种具体情况，如图 25-13 所示。

因 素	描 述
系统的技术质量	如果报告说有严重的系统缺陷影响了很多用户对系统的使用，那么很有必要发放一个缺陷修补版本。然而，要是小的缺陷，就可以通过发放一个补丁的办法（一般是放在因特网上）来让用户自己对当前系统执行修补操作
平台改变	当操作系统平台的新版本发布时，必须创建软件应用的一个新发布版本
Lehman 的第五条定律（见第 9 章）	他提出在每个版本中的功能增量基本上是个定数。因此，如果一个系统版本具有相当多的新功能，他就会不得不跟随一个修补版本
竞争	由于竞争对手的相应产品的出现，发布一个新的系统版本是必要的
市场需求	机构的市场部可能已经承诺在特定日期发布一个新版本
客户变更建议	对于定制的系统，客户可能已经提出了一些特别的系统变更建议，并已经为此支付的费用。他们希望系统一实现就发布

图 25-13 影响系统发布规划的因素

系统的发布版本不仅仅是这个系统的可执行代码。发布版本还包括：

- 1. 配置文件，定义对于特定安装，发布版本应该如何配置。
- 2. 数据文件，比如错误信息的文件，是成功进行系统操作所必需的。
- 3. 安装程序，用来帮助在目标硬件上安装系统。

4. 电子和书面文档，用于系统说明。
5. 包装和相关的宣传，为发布版本所做的工作。

发布版本的创建是创建包含系统发布版本的所有组件在内的文件和文档集合的过程。

程序的可执行代码及所有相关数据文件都必须在版本管理系统中找到，并贴上发布标识符。对不同的硬件和操作系统要写出配置描述。针对需要配置自己的系统的客户，也要为其准备好使用说明书。如果发布了可在机器上阅读的手册，其电子拷贝必须与软件一起保存起来。还要写出安装程序的脚本。最后，当所有的信息齐备的时候，必须准备该软件的可执行母版移交给客户或者销售渠道。

计划新系统发布版本安装的时候，发布版本管理者不能想当然地认为客户总是想安装新的系统版本。一些系统用户可能满意于已有的系统版本。也许他们认为新版本不值得为之付费。因此系统的新版本不能依靠以前的版本的安装。为阐明该问题我们看下面的情景：

1. 系统的发布版本 1 已发布并投入使用。

2. 发布版本 2 需要安装新的数据文件，而有些客户不需要发布版本 2 所提供的设施，因此仍然保留发布版本 1。

3. 发布版本 3 需要发布版本 2 中安装的数据文件，没有自己新的数据文件。

软件发布人员不能认为发布版本 3 所需要的文件已经安装到所有的地点。有些地点可能从发布版本 1 直接到发布版本 3，跳过了发布版本 2。有些地点则可能已经根据具体情况对与发布版本 2 有关的数据文件做了修改。因此数据文件必须随同系统的发布版本 3 一起发布和安装。

由于发布软件产品的新版本会有很高的销售和包装成本，所以通常卖主们仅为新平台或在加入重要的新功能时才进行发布。然后，他们要向用户收取新软件的费用。若已发布的软件出现错误，软件卖主会在网上发布补丁来修补已存在的软件，用户可以从网上下载。

使用下载的补丁的问题是：许多客户可能永远也发现不了这些问题补丁的存在，而且不能理解为什么需要安装。他们只好继续使用旧的有缺陷的版本，这是存在威胁他们业务的隐患的。某些情况下，补丁是为修补系统的安全漏洞设计的，安装补丁失败就意味着系统容易受到外部攻击。为了避免这些问题，大众软件卖主，比如 Adobe、Apple 和微软，通常实现自动更新，在有新的小型发布的时候系统自动更新。然而，对于定制系统来说，这不起作用。因为这些系统并不存在于针对所有客户的标准版本中。

要点

- 配置管理是对开发软件系统所做的管理。在维护一个系统时，配置管理团队的作用是保证系统的变更是在受控状态下完成的，并且详细记录已经实现的改变。
- 主要配置管理过程包括变更管理、版本管理、系统构建以及发布管理。存在能够支持所有这些过程的软件工具。
- 变更管理包括评估来自系统客户和其他信息持有者的变更提议，并判断在系统新版本中实现这些变更的费用是否合适。
- 版本管理包括追踪软件组件由于变更而产生的不同版本。
- 系统构建是将系统组件装配成一个在目标计算机系统上的可执行程序的过程。
- 软件应该频繁地重新构建，在新版本构建之后立即进行测试。这样可以更容易察觉最后一次构建引入的错误和问题
- 系统的发布包括可执行代码、数据文件、配置文件和文档。发布管理包括决定何时发布一个系统，准备好所有待发布的信息，并为每一个系统发布版本编制好文档。

进一步阅读材料

《Configuration Management Principles and practice》这本书通俗易懂，不仅包含配置管理的标准和惯常方法，还含有更适合现代快速软件开发过程的方法（Anne Mette Jonassen Hass, 2002, Addison-Wesley）。

《Software Configuration Management Patterns: Effective Teamwork, Practical Integration》这是一本相对较短，容易阅读的书，在配制管理实践方面尤其在对敏捷开发方法，给了很好的具体建议（S. P. Berczuk 和 B. Appleton, Addison-Wesley, 2003）。

《High-level Best Practices in Software Configuration Management》，这是篇网络文章，由一位配置管理工具提供商的员工所写，是对于软件配置管理优秀行为的很好的入门介绍（L. Wingerd and C. Seiwald, 2006）。<http://www.perforce.com/perforce/papers/bestpractices.html>。

《Agile Configuration Management for Large Organizations》这篇网络文章描述能在敏捷开发过程中使用的配置管理实践，特别强调了如何将这些实践扩展到大项目和大公司（P. Schuh, 2007）。<http://www.ibm.com/developerworks/rational/library/maro7/schuh/index.html>。

练习

- 25.1 如果一个公司没有制定有效的配置管理政策和过程，列举5种可能出现的问题。
- 25.2 在变更管理过程中使用变更请求表格作为核心文件的好处是什么？
- 25.3 列举支持变更管理过程工具应该包含的6种特征。
- 25.4 解释组件的每个版本都必须唯一标识的原因。评述使用简单的基于版本号的版本标识模式的问题。
- 25.5 设想如果两个开发人员同时修改3个不同组件，当他们想要合并他们所做出得修改时，可能出现什么问题？
- 25.6 开发软件的团队现在越来越多地在不同地点工作。列举一些能够支持分布软件开发的版本管理系统的特征。
- 25.7 列举在系统构建中可能碰到的问题。当在主机上为目标机器构建系统时可能会发生哪些特殊问题？
- 25.8 关于系统构建，解释一下为什么有时有必要维护已经过时的计算机，假设曾经在该计算机上开发过大型软件系统？
- 25.9 在系统构建中一个常见的问题是将物理文件名包括在系统代码中，而文件名指出的文件结构与目标机器上的文件结构不一致。试写出一组程序设计者指南以帮助他们避免这个问题，以及你能够想到的其他系统构建中容易发生的问题。
- 25.10 在建立一个大型软件系统的发布版本的过程中，工程人员必须考虑哪5个因素？

参考书目

- Ahern, D. M., Clouse, A. and Turner, R. (2001). *CMMI Distilled*. Reading, Mass.: Addison-Wesley.
- Bamford, R. and Deibler, W. J. (2003). 'ISO 9001:2000 for Software and Systems Providers: An Engineering Approach'. Boca Raton, FL: CRC Press.
- Paulk, M. C., Weber, C. V., Curtis, B. and Chrissis, M. B. (1995). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Mass.: Addison-Wesley.
- Peach, R. W. (1996). *The ISO 9000 Handbook, 3rd edition*. New York: Irwin Professional Pub.
- Pilato, C. M., Collins-Sussman, B. and Fitzpatrick, B. W. (2004). *Version Control with Subversion*. Sebastopol, Calif.: O'Reilly Media Inc.
- Vesperman, J. (2003). *Essential CVS*. Sebastopol, Calif.: O'Reilly and Associates.

过程改善

目标

本章的目标是介绍软件过程改善，这是一种提升软件质量和减少开发成本的方式。读完本章，我们将了解以下内容：

- 了解有关将软件过程改善作为提升产品质量以及软件过程的效率和效果的手段的基本原理；
- 了解软件过程改善的基本原理以及过程改善的循环过程；
- 学会如何使用目标 - 问题 - 量度方法指导过程度量；
- 介绍过程能力和过程成熟度的概念，以及由 SEI 所提出的过程改善的 CMMI 模型的一般形式。

如今，工业界一直都有尽可能地获得更便宜、更好的软件的需求，且要求软件在日益严格的最后期限前交付。所以大多数软件公司转向通过改善软件过程来提高产品质量，降低软件开发成本，及加速它们的开发过程。过程改善就是要充分了解现有的过程并改变这些过程，实现提高产品质量及降低成本和减少开发时间的目的。

过程改善和变更使用两种截然不同的方法：

1. 过程成熟度方法，重点改善过程和项目管理以及将优良的软件工程实践引进到机构中。过程成熟水平反映了在机构的软件开发过程中采用的优良的技术和管理实践的程度。此方法的主要目标是改善产品质量和过程可预测性。

2. 敏捷方法，重点是迭代开发和减少软件开发过程的费用。敏捷方法的主要特点是功能的快速交付和对客户需求变更的快速反应。

这两种方法各自的拥护者都对另一种方法的益处持怀疑态度。过程成熟度方法植根于计划驱动开发，因为增加了和编程不直接相关的活动，通常需要增加费用。敏捷方法专注于正在开发的代码并且刻意使流程和文件最少。

第3章和本书其他一些地方已经讨论过敏捷方法，所以本章重点介绍过程管理和基于成熟度的过程改善方法。这并不代表作者更钟情于此方法而不提倡敏捷方法。实际上，对于小型和中型项目，采用敏捷方法可能是最节省成本的过程改善策略。但是对于大型系统、要求极高的系统以及由在不同公司的开发人员所开发的系统，管理问题通常是项目遭遇问题的原因。对于业务庞大的公司和复杂系统工程，改善过程应该考虑基于成熟度的方法。

正如第24章所述，创建软件系统所用的开发过程影响系统的质量。因此，许多人相信改善软件开发过程能提升软件质量。过程改善的这些观点起源于一个叫 W. E. Deming 的美国工程师，他于第二次世界大战后在日本的工业界工作，主要从事帮助改善质量的工作。日本的工业采用连续性过程改善（在日语中称为 kaizen）过程已经有很多年了。该过程对日本产品的总体高质量的贡献是巨大的。

Deming（还有其他人）提出了统计质量控制的观点。它是通过对产品缺陷的数目进行度量并将之与生产过程相关联。其目标是通过分析和修改过程使引入缺陷的几率减少并改善缺陷检测。一旦缺陷统计数目降低，就对过程进行标准化并开始新一轮改善活动。

Humphrey (1988) 在他的过程控制的书中写道, 在软件工程中可以使用相同的技术。他指出:

W. E. Deming 于第二次世界大战后在日本工业界工作, 在工业中使用统计过程控制的概念。虽然软件和工业产品存在着很大的不同, 这个概念在软件中使用, 会像它在汽车、照相机、手表以及钢铁中一样。

虽然它们有很明显的相同点, 作者不赞同 Humphrey 所说的能把来自制造工程的结果轻松转移到软件工程中的观点。在制造业中, 过程和产品之间的关系是非常明显的。制造通常包括准备自动工具和产品检测过程。如果有人在校准机器的时候出错, 这会影响这台机器生产的所有产品。避免在准备机器的时候发生错误以及引入更有效的检测过程可以明显改善产品质量。

当产品是无形的, 或者在某种程度上依靠智力活动而难以自动化时, 这种过程质量/产品质量之间的关系就不那么明显了。软件质量不仅受制造过程影响, 更受到设计过程影响, 个人技能和经验非常重要。有时候, 所用的过程对产品质量而言是最重要的决定因素。但并非都是如此, 对于有创新的应用来说, 参与到过程中的人员所起的作用要比过程重要得多。

对于软件产品 (或其他智力产品, 如书籍、电影等, 其质量主要依赖于设计) 而言, 有 4 个因素能够影响产品质量, 如图 26-1 所示。

每种因素的影响程度与项目的类型和规模相关。对于由多个独立的子系统组成的大型系统来说, 一般是由多个团队开发的, 产品质量的决定性因素来自于软件过程。大型项目的主要问题是集成、项目管理和沟通。在这样的团队中, 通常人员的能力和经历是参差不齐的, 项目工期可能有数年之久, 在这么长的时间里, 甚至到最后一个人也“老”人也留下来。项目的生命周期中开发团队的人员可能发生完全的变动。

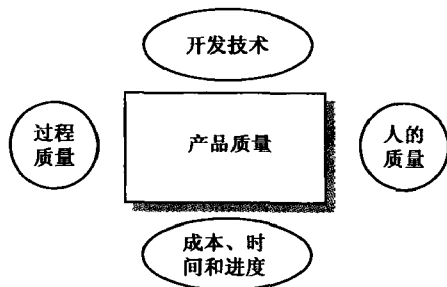


图 26-1 影响软件产品的因素

然而, 对于只有几个团队成员的小项目来说, 开发团队的人员素质就会比所用的开发过程更重要了。所以, 敏捷方法宣扬人的重要性大于使用的开发过程的重要性。如果团队有高水平的能力和经验, 产品的质量也就会高, 和使用的过程无关。如果团队成员缺乏经验也不熟练, 一个好的过程可能会限制灾害的发生, 但是单凭过程本身无法带来高品质的软件。

对于小团队, 好的开发技术就变得特别重要。团队不能在琐碎的管理事务中耗费太多的精力。工程人员把大部分时间用在系统设计和编程上, 所以好的工具将极大地影响他们的劳动生产率。对于大型项目, 一些基本的开发技术对信息管理是必要的, 而复杂的软件工具就相对不太重要了。团队成员花相对较少的时间用于开发活动, 较多的时间用在沟通和了解系统其他部分上了。但是, 支持沟通的 Web 2.0 工具, 比如 Wikis 和博客, 能极大改善分散的团队之间的沟通情况。

一个项目, 不管它的人员、过程或者工具因素, 若预算偏低, 或者交付进度计划不现实, 产品质量将受到影响。一个好的过程要得到有效的实施需要配备适当的资源。如果资源不充足, 过程就不可能有效地进行。如果资源配备不适当, 那就只有极优秀的人员才能使项目顺利完成了。要是项目赤字严重, 即使再优秀的人也无法保证产品质量不受影响。如果没有足够的开发时间, 交付的软件可能会减少功能或者降低可靠性或性能水平。

通常软件质量存在问题的真正原因不是管理不善、过程不恰当或质量培训差, 而是源于机构受到的生存压力。开发机构为了赢得开发机会, 故意压低项目预算或承诺快速交付系统。为了实现这些承诺, 开发机构就会制定不切实际的开发进度表。结果是软件质量受到不利影响。

26.1 过程改善过程

第2章介绍了软件过程是一个活动序列的总的思想，该序列的执行结果产生一个软件系统。前面曾介绍过通用过程，比如瀑布模型和基于复用的开发，也讨论过最重要的过程活动。机构可以通过对这些通用过程实例化为自己的开发软件的特殊过程。

在所有的机构中我们都能看到软件过程的存在，从一个人的公司到一个大的跨国公司。这些过程可以根据过程的形式化程度、所开发的产品的类型、机构的规模等，划分为不同的类型。不存在适用于所有开发机构或者适应于某一特定类型的所有软件产品的“理想的”或者“标准的”软件过程。每个公司必须根据其规模、背景、员工的技能、开发的软件的类型、客户和市场需求以及公司的文化开发自己的过程。

因此，过程改善不仅仅意味着采用特别的方法或工具，或者使用一个公开的、通用的过程。虽然开发机构在开发相同类型的软件时明显地有很多相同之处，但总是有许多机构的自身因素、规程和标准在影响着过程。如果仅仅简单地试图将某个过程转变为一个通用的过程，引入过程改善是不会成功的。必须总体考虑到本地环境和文化，以及过程变更提议所带来的影响。

我们还必须考虑我们想要改善过程的哪些方面。目标可能是改善软件质量，因而我们可能希望引入新的过程活动，以改变软件开发和测试的方式。也有可能是想改善过程本身的一些属性，这就必须判别何种过程属性对公司是最重要的。图26-2列出了可以作为改善目标的过程属性的一些例子。

过程特性	描述
易懂性	过程在多大程度上做了明确的定义，对过程定义理解的难易如何
可视性	过程活动是否有一个清晰的结果，从而使过程的进展能从外部观察到
支持性	过程活动在多大程度上得到CASE工具的支持
可接受性	所定义的过程对于从事软件产品开发的工程人员来说是容易接受的和可用的吗
可靠性	所设计的过程中的错误是否能在其导致产品错误之前得到避免或者被捕获
鲁棒性	过程是否能在无法预料到的问题发生时继续工作
可维护性	过程的进化是否能反映机构需求的变更或者识别出过程改善之处
快速性	从给定系统描述到移交系统的过程有多长

图 26-2 过程属性

很明显这些属性是相关联的，有时候是正相关，而有时候是负相关。因此，可见性得分高的过程可能也是容易理解的。过程观察员能够根据产生的输出推断活动的存在。另一方面，过程可见性和速度成反比。只有相关人员产生关于过程本身的信息才能使过程可见。由于生产这些文件要花费时间，这可能会减慢软件的生产速度。

同时优化所有的过程属性进行过程改善是不可能的。例如，如果目标是快速的开发过程，那么就必须降低过程的可见性。如果想要增加过程的可维护性，就必须采用能反映更广泛的机构实践的、且在公司不同部门使用的流程和工具。这可能降低过程的局部可接受性。工程人员可能引入了自己的局部流程和非标准工具支持他们的工作方式。因为这些流程和工具富有成效，他们可能并不愿意放弃这些而支持标准的过程。

如图26-3所示，过程改善是一个循环的过程，它包含3

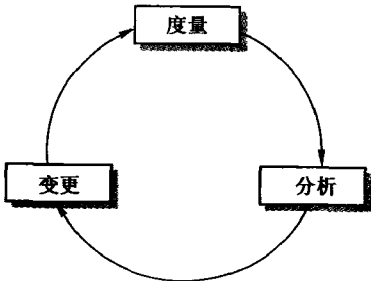


图 26-3 过程改善周期

个子过程：

1. **过程度量** 对当前项目或产品的属性的度量。它的目的是根据机构对过程改善的目标来改善这些尺度。这形成了一个基线，能够帮助判断过程改善是否有效。

2. **过程分析** 对当前的过程进行评估，找出过程的缺点和瓶颈。在这一阶段经常建立描述所要开发的过程的过程模型（有时叫过程地图）。分析可能重点考虑过程特征，比如快速性和鲁棒性。

3. **过程变更** 对已经发现的过程弱点提出解决问题的过程变更提议。执行这些变更并重新开始收集有关变更有效性的数据。

没有具体的有关过程的或有关使用这个过程进行开发的软件的数据，就不可能对过程改善的价值做出评估。然而，启动过程改善过程的公司可能没有作为改善基线的可用的过程数据。因此，作为变更的第一个周期的一部分，必须引入过程活动去收集关于软件过程的数据和评测软件产品的性质。

过程改善是一项长期的活动，改善过程中的每一个阶段都可能持续几个月。同时过程改善也是连续性的活动，不管引入何种新的过程，业务环境都会发生改变，新的过程本身必须不断进化从而将这些变更纳入考虑之中。

26.2 过程度量

过程度量是软件过程的量化数据，比如用于执行过程活动的时间。这样的例子是，我们计算一下开发程序测试案例所需要的时间。Humphrey（1989）在他关于过程改善的书中认为，过程度量和产品特性对于过程改善是非常关键的。Humphrey 也提出度量在小规模、个人过程改善中发挥着重要作用（Humphrey，1995），因为个人努力变得更有生产率。

过程度量可以用于评估是否效率已经得到了改善。例如，可以对用于测试的工作量和时间进行监控。对测试过程的有效改善应该能减少工作量和/或测试时间。不过，过程度量本身不能够用于确定产品质量是否有所改善。产品质量数据（见第 24 章）也必须要收集并与过程活动相关联。

能够收集的过程量度数据有 3 类：

1. **完成某一特定的过程所花的时间** 这个时间可以是过程总的耗费时间，也可以是日历时间，或者是个别工程人员花在过程中的时间，等等。

2. **某个特定过程所需要的资源** 资源可以包括以人日计算的总的工作量、差旅费或者是计算机资源等。

3. **某个特定事件发生的次数** 可以监控到的事件发生频度的例子包括：在代码检查时发现的缺陷数量，需求变更请求数量，由于需求变更而相应修改的程序代码行数等。

前两种度量类型可以用于发现过程变更是否提高了过程的效率。比如，在软件开发过程中确定几个点，如接受需求、体系结构设计完成、完成测试数据生成等。我们就可以度量从一个点到另一个点之间需要花费的时间和工作量。度量得到的数据可以指示出过程的哪些地方需要改善。在引入变更之后，从对系统属性的度量中能看出过程变更是否成功减少了需要的时间和成本。

事件发生数量的度量对软件质量能产生更直接的影响。举例来说，通过改变程序审查过程发现了更多的缺陷，修复这些缺陷必然使产品质量得以改善。当然这需要得到后续的产品度量的证实。

在过程度量中的一个主要困难是弄清楚应该收集关于过程的哪些信息以支持过程改善。Basili 和 Rombach（1988）提出了他们称之为 GQM（目标 - 问题 - 量度）的范式，广泛应用于软件

和过程度量。Basili 和 Green (1993) 描述了如何在美国航空航天局 (NASA) 中一个长期的基于度量的过程改善项目中使用这个方法。

在过程改善中使用 GQM 范式 (见图 26-4) 帮助回答 3 个关键问题:

1. 为什么引入过程改善?
2. 需要哪些信息用于识别和评估改善?
3. 需要哪些过程和产品的度量以提供这些信息?

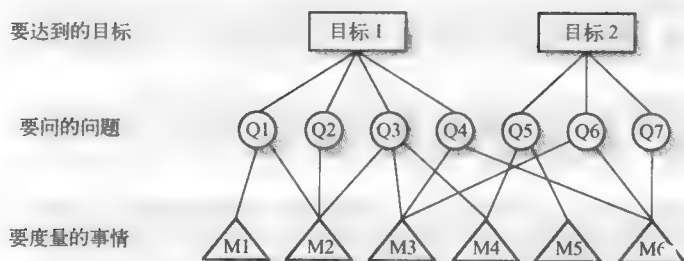


图 26-4 GQM 范式

这些问题直接和 GQM 范式中的抽象概念 (目标、问题、量度) 有关:

1. **目标** 机构想要达到什么目标。这不一定直接与过程属性有关, 而是与过程如何影响产品或机构本身有关。目标的例子可能是: 一个过程成熟度的改善的水平 (见 26.5 节), 更短的产品开发时间, 增加的产品可靠性等。

2. **问题** 这些是对目标的进一步提炼, 找出目标相关的特定范围的一些不确定性。一般而言, 一个目标会有许多相关联的问题需要我们回答。和缩短产品开发时间这个目标有关的问题例子可能是: “当前过程的瓶颈是什么?”, “如何缩短与客户商定产品需求的时间?”, “我们的测试中有多少能有效地发现产品缺陷?”

3. **量度** 这些是需要收集的度量数据以帮助回答问题, 并帮助确定过程改善是否达到了预期的目标。为了帮助回答上面的问题, 我们可能收集的量度有: 完成各个过程活动所需要的时间 (规范化为特定的系统大小所需要的时间), 在客户和承包商之间为每个需求变更所做的正式沟通的次数、每次测试发现的缺陷数。

GQM 方法用于过程改善的好处是, 它将机构关注的焦点 (目标) 和特别过程关注的焦点 (问题) 相分离。该方法为决定应收集哪些数据提供了基础, 并建议根据希望回答的问题用不同的方式分析收集到的数据。

GQM 方法已经得到了一定的发展并与 SEI 的能力成熟度模型 (Paulk 等, 1995) 一起结合在软件过程改善的 AMI (分析、度量、改善) 方法中。AMI 方法的开发者提出了分阶段的过程改善方法, 在机构对过程引入标准化之后再开始度量, 而不是马上开始度量。AMI 手册 (Pulford 等, 1996) 提供了实现基于度量的过程改善的一些指导原则和实践忠告。

如第 24 章中所提到的, 解释度量的确切含义有时是个问题。例如, 有个软件已经交付给外部进行测试, 我们要度量用于修复该软件中报告出来的错误的平均时间。这是介于团队接收到错误报告的时刻和正式地对这个报告标记为“已清除”的时刻之间的一段时间。如果引入新的基于 Web 的工具进行错误报告, 在使用一段此工具之后, 将会发现修复报告的错误所用时间在减少。

你可能因而就断言引入错误报告工具减少了修复错误的时间。当观察到这个量度的变化时, 总是很容易将这些变化归结于所引入的过程变更。但是, 对改善做出过于简单化的假设是危险

的。量度的变化可能是由完全不同的因素引起的，比如项目组人员的变化、项目进度的变化或者管理的变化。也可能是这样的情况，即团队的做法发生了简单的改变，因为他们的工作正在被度量。在错误报告工具这个例子中，观察到的变化可能会有以下一些原因：



过程实践的分析

过程分析的一个方法是使用调查问卷去发现在多大程度上好的软件工程实践在使用。因而，对过程中的某些阶段，例如需求工程，你可以识别在公司中对于正在开发的系统类型哪种实践是最为合适的，并提问关于这些实践在多大范围内得到了使用。这是在作者的关于需求工程过程改善的书中所建议的方法（Sommerville 和 Sawyer, 1997）。

<http://www.SoftwareEngineering-9.com/Web/Prolmp/goodpractice.html>

1. 新系统可能减少了开销，因此有更多的时间用于修复错误。这带来了平均“错误修复”时间。过程改善带来了真的不同。
2. 新系统可能不能对用于修复错误所需时间带来改观，但是却使记录信息更加容易。因此，使用新系统能更准确地度量错误修复时间。修复错误的平均时间实际并未变化。
3. 在引入新系统之前的度量可能是部分通过对系统的测试得到的。容易发现并能够快速修复的错误已经被修复了，只有“难修复的错误”还存在并需要花费更长的时间去修补。但是，引入错误报告系统之后，度量是在对新系统测试之初做出的，修复的错误是易于修复的“简单错误”。
4. 测试团队的新管理人员可能指示团队成员将用户界面的不一致性作为错误来报告，而不是像先前那样被忽略掉。这表示将报告更多能够快速修复的“简单错误”。

度量是过程和过程变更的证据生成的方式。但是必须在确定过程变更有效之前，加上其他关于过程的信息一起阐述这些证据。但是这些证据需要和过程的其他信息结合起来加以解释，然后我们才能相信过程变更的有效性。我们应该总是将度量和变更的定性评估一起使用。这包括和参与过程的人员交谈，听取他们对于这些变更有效性的看法。这不但揭示了可能影响过程的其他因素，还揭示了开发团队在多大程度上接受所提议的变更，以及这些是如何影响实际开发行为的。

26.3 过程分析

过程分析是指对过程的研究，帮助我们理解过程的关键特征以及相关人员在实际中是如何完成这些过程的。图 26-3 中提出在过程度量之后进行过程分析。这是一个简化说法，因为在现实中，这些活动是交织在一起的。你需要进行分析来了解要度量什么和什么时候进行度量。我们将不可避免地要对要度量的过程进行更深层的了解。

过程分析有许多密切相关的目标：

1. 理解过程中的活动以及这些活动之间的关系。
2. 理解过程活动之间的关系以及理解所做出的度量。
3. 将特殊过程或者是你正在分析的过程和机构中其他地方的差不多的过程或者同类型的一个理想化的过程联系起来。

在过程分析中，我们试图去理解过程中发生了什么。我们要去寻找有关过程的问题和效率低下的线索。我们也应该关心这个过程的使用范围、用于支持过程的软件工具以及过程是如何受机构约束影响的。图 26-5 列出了过程分析阶段可能调查的过程的一些方面。

过程方面	问 题
采用和标准化	是否在整个机构中对过程进行了文档化和标准化？如果没有，是否意味着做出的任何度量仅仅针对某个具体的过程实例？如果过程没有标准化，那么对一个过程做出的变更就是不能转移到公司中其他地方相似的过程中的
软件工程实践	是否有些著名的好的软件工程实践没有包含在过程中？为什么没有包含进来？缺少这些实践影响产品特性（比如交付的软件系统中的缺陷数）吗
机构约束	影响过程设计和过程执行方式的机构约束是什么？例如，如果过程包括处理分类的材料，过程中可能有活动是检查有的分类信息由于要发布给外部机构因而没有包括在任何材料中。机构的约束会意味着有些过程变更不能进行
沟通	过程中如何管理沟通？沟通问题和引入的过程度量是如何关联的？沟通问题是许多过程的主要问题，沟通的瓶颈经常也是项目延迟的原因
内省	过程是深思熟虑的吗（比如，参与过程的角色是否明确地思考并且讨论过程，以及如何改善的）？有没有机制让过程的参与者能够建议过程改善
学习	人员是如何在开发团队中学习所使用的软件过程？公司是否有过程手册和过程培训项目
工具支持	软件工具支持哪些方面和不支持过程的哪些方面？对于不支持的地方，有没有能够以低成本部署的工具对此提供支持？对于支持的地方，工具是起作用的和高效的吗？有更好的工具吗

图 26-5 过程分析的各个方面

最常用的过程分析技术有：

1. **调查问卷和会谈** 向项目中的工程人员和管理人员询问实际过程中执行的情况。在对工程人员单独会谈时对调查问卷中的回答再做进一步细化。正如下面要讨论的，讨论可以组织成围绕着软件过程模型来进行。

2. **深入实际调查研究** 深入实际调查研究（见第4章）是到现场观察过程参与者的具体工作，这可以用来了解软件开发作为一项人的活动的一些本质。如此分析可以揭示无法用调查问卷和会谈方式得到的过程的微妙和复杂之处。

每一种方法都有其优点和缺点。基于调查问卷的分析在找出恰当的问题后很快就会完成。不过，如果问题晦涩难懂或问题本身不恰当，我们将得到一个不完全的或错误的模型。此外，基于调查问卷的分析对参与者来说看起来像是一种评估或评价。被询问的工程师可能会认为询问者并不是真的想知道过程中真实的一面，所以回答时难免有投其所好的可能。

约见参与过程的人将比调查问卷更加开放。可以准备一系列的问题而根据不同人的反馈来适当做出调整。如果给参与者机会去讨论更宽泛的问题，我们将发现过程参与者乐于谈论过程的实际问题，以及在实际中过程是如何被改变的，等等。

几乎在所有过程中，参与的人员都会做出局部的变更使过程适应本地环境。深入实际调查研究比会谈更有可能发现使用的真实的过程。不过，这是一个很费时间和金钱的活动，往往需要持续数月。它依赖于对过程的外部观察。一个完整的分析必须贯穿项目初始阶段至产品交付和维护阶段整个过程。对于大型项目，可能持续很多年，很明显针对完整的过程，深入实际调查分析就不实用。深入实际调查研究的分析方法在某个过程片断需要深入了解的情况下是最有用的。一旦从访问材料中判断出需要进一步调查的区域，然后做集中深入实际调查发现过程的细节。

在分析过程的时候，一般先从定义过程中的活动以及这些活动的输入输出的过程模型开始。模型可能也包括过程角色信息（即负责执行活动的人员或角色），以及必须产生的关键的交付文档。可以使用非正式的标记描述过程模型，也可以使用更正式的表格形式的标记方法，UML活

动图，或者是业务过程建模标记，比如 BPMN（见第 19 章）。本书中使用了大量的过程模型的实例，用于描述软件过程。

过程模型是聚焦在过程中的活动以及活动间信息转换的好方法。这些过程模型不必是形式化或者完整的，它们的目的是引起讨论而不是详细记录过程。和参与过程相关的人员的讨论以及对过程的观察通常组织成围绕正式过程模型的一组问题去展开。下面是这些问题的一些例子：

1. 有哪些活动在模型中没有，但在实际中发生？模型不可避免是不完整的，但是不同的人定义不同的缺少的活动，这表示机构当中并未连续执行过程。
2. 模型中的过程活动是否有我们认为效率低下的？是如何效率低下的，如何才能改进？这些效率低下的活动是如何影响已做出的过程度量的？
3. 出现问题时会发生什么？团队是否继续遵守模型中定义的过程，还是放弃这些过程并采取紧急行动？如果放弃这些过程，这表明软件工程师不相信过程足够好或者过程没有足够的灵活性以处理意外情况。
4. 过程中各个不同阶段有些什么角色？他们如何交流？信息交流的时候通常会有什么瓶颈？
5. 模型中显示的活动使用什么工具支持？它们是有用的并普遍使用的吗？如何改善工具支持？

完成软件过程分析之后，就会对过程以及将来过程改善的可能性有更深刻的理解。也应该理解过程改善的约束以及其是如何限制可能引进的改善的范围。

过程异常

软件过程是非常复杂的。在机构中定义好了的过程模型只意味着过程的一种理想状况，而没有考虑任何意外情况。事实上，项目管理者每天都会遇到一些意想不到的问题。一旦找到这些问题的解决办法，就必须动态地修改这个“理想”的过程模型，以解决这些发现的问题。一个项目管理者必须应对的异常类型有以下几种：

1. 几个主要成员同时生病，并恰巧在关键项目复查之际。
2. 计算机信息安全遭到严重破坏，这意味着外部通信无法正常工作达数天。
3. 公司机构改组使管理者必须忙于应付机构事务，因而影响了项目的管理。
4. 有未预料到的新的项目投标请求，一部分精力必须从当前项目转移到这个项目投标工作中。

一般来说，异常的发生总是会以某种方式影响和改变项目的资源、预算或进度。很难预测所有的异常，因而无法将其统统考虑到形式化过程模型中去。这就需要项目管理者能动态改变标准过程来应对这些未预料到的客观环境。



软件过程建模

软件过程建模开始于 20 世纪 80 年代早期（Osterweil, 1987），其长期目标是用过程模型作为组织和协调的支持工具。过程模型应该包含关于过程活动的信息、输入输出、过程中的角色等。人们研究了多种针对特殊目的的软件过程建模符号系统，但是它们基本都被像 BPMN 或 UML 活动模型这样的业务过程建模符号系统所取代了。

<http://www.SoftwareEngineering-9.com/Web/Proclmp/spm.html>

26.4 过程变更

过程变更是指对已经存在的过程进行修改。正如前面谈到，我们可以通过引入新的实

践、方法或工具，改变过程活动的顺序，从过程中导出或去除可交付内容，改善沟通或使用新的角色和责任等方法做到这一点。我们需要为过程改善设定目标，像“将集成测试中发现的错误减少25%”，这可以驱动过程变更。在变更实现后，使用过程度量来评估变更的有效性。

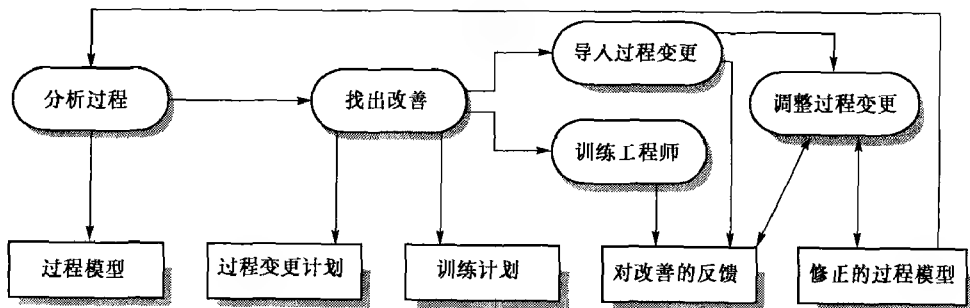


图 26-6 过程变更的过程

在过程变更过程中（见图 26-6），有 5 个关键的步骤：

1. **改善识别** 这个阶段所关心的是使用过程分析的结果来识别解决一系列问题的方法，包括质量问题、进度瓶颈问题，或成本效率问题，这些是在过程分析中确定的问题。通过提出新的过程、过程结构，方法和工具来解决过程问题。比如，某个公司认为许多软件问题源于需求问题。使用需求工程最佳行为指导（Sommerville 和 Sawyer, 1997），可以找出各种可能引入或改变的需求工程实践。

2. **改善优先级排序** 这个阶段要得到可能的变更，然后决定它们在执行过程中的优先级。当有很多变更被识别出的时候，通常不能一次把它们都引入；必须决定哪个更重要。我们可能基于需要变更的特定的过程领域、引入变更的费用、变更影响的机构和其他因素来决定它们的优先级。比如，某公司可能考虑将引入需求管理过程去管理进化的需求作为最高优先级过程变更。

3. **过程变更引入** 引入过程变更就是要加入新的规程、方法和工具，将它们集成进其他的过程活动。在这一过程中有两点非常重要，一是要有充足的时间，二是要保证这些变更不和其他的过程活动以及机构已有的规程和标准发生冲突。这可能包括学会使用需求管理工具以及设计过程来使用这些工具。

4. **过程变更的培训** 如果不进行培训，就不可能得到来自过程变更的全部好处。相关工程人员需要理解提出的变更以及如何执行新的变更后的过程。机构往往忽略对过程变更给予充分的培训并急于进行，结果变更带来的不是产品质量的提高而是变得更差。在需求管理的情况下，培训可能包括：讨论需求管理的价值，过程活动的解释，对已选择的工具的介绍。

5. **变更微调** 变更不可能一引入就立竿见影。需要有一个调整阶段，其间会有一些小的问题出现，也会有一个对过程不断修改并执行的迭代过程。这个调整阶段应该持续几个月的时间，直到开发工程师能对新过程愉悦接受为止。

通常同时引入太多的变更是不明智的。引入太多变更除了会带来培训的困难外，还难以评估每个变更的效果。过程变更一经引入，通过进一步的分析来识别过程中的问题，提出改进等，改善过程能反复地进行。

除了评估变更后的过程有效性的困难外，还有如下两个过程变更必须面对的主要困难：

1. **拒绝变更** 团队成员或者项目管理者可能拒绝引入过程变更，提出变更不起作用的理由，

或者延迟引入变更。有些情况下，他们故意妨碍过程变更并且举出数据说明提议的过程变更无效。

2. 变更坚持 虽然可能在开始阶段引入过程变更，但往往在短时间后就会放弃过程革新，过程回复到之前的状态。

拒绝变更的人员可能来自变更过程涉及的项目管理者和工程人员。项目管理人员通常拒绝过程变更是因为任何变革都带有未知的风险。过程变更的目标可能是加快软件的生产率或者减少软件的缺陷。但是这些过程变更总是有无效的风险，或者引进变更需要的时间比节省的时间还要长。对项目管理人员的评判是根据项目是否能按时并按预算生产软件。因此，管理人员更喜欢效率低但可预测的过程，而不喜欢虽然能给机构带来益处但可能也会带来短期风险的优化过程。

工程人员因为类似原因或者因为他们将这些过程视为对他们专业性的威胁也可能拒绝引入新过程。也就是说，他们可能认为新的预定义过程使他们失去更多的自主性，并且使得他们的技能和经验的价值受到低估。他们可能认为新过程将意味着需要的人员减少，他们可能失去工作。工程人员因此不愿意学习新技能、工具或者工作方式。

作为一个管理者，要善于观察所在团队中队员的感受。在变更过程中，我们必须使用各种方法来激励我们的队员，了解他们的顾虑，鼓励他们参与计划新的过程。通过让他们成为过程变更的信息持有者，我们会发现他们更愿意让变更进行下去。业务过程再工程（Hammer, 1990; Ould, 1995）是 20 世纪 90 年代的一个潮流，它要引入彻底的过程变更，最后以完全的失败而告终，因为它没有将所涉及的人的正当关注考虑在内。

管理者会担心过程变更会给项目进度和成本带来不利影响，为了解决这种忧虑，必须增加项目预算用于变更引起的额外开销和延迟。必须以现实态度对待变更的短期益处。变更不可能带来大范围的、立即的改善。过程变更的好处是长期的，而不是短期的。所以必须在多个项目中共同支持过程变更。

引入变更不久后便放弃是个常见问题。提出变更的人坚信变更会带来改善。他会努力工作保证变更有效，人们能接受新的过程。然而，一旦“坚信者”离开，他的位置可能被某些不那么愿意把力气花在新过程上的人所替代。相关人员可能因此回复到之前的工作方式。如果引进的过程还未普遍采用，过程变更的所有好处还未完全显现时，这特别容易发生。

为了克服变更难于坚持的问题，CMMI 模型（将在 26.5 节中讨论）强烈建议过程变更制度化。这表示过程变更不依赖于个人，变更成为了公司标准实践的一部分，并获得公司范围的支持和培训。

26.5 CMMI 过程改善框架

美国的软件工程学会（SEI）是一个致力于改善美国软件工业能力的机构。在 20 世纪 80 年代中期，SEI 启动了对承包商能力评估方法的研究。他们特别关注由国防部出资的软件项目的承包商。该项能力评估研究工作的成果是 SEI 的软件能力成熟度模型（CMM）（Paulk 等，1993; Paulk 等，1995）。该模型轰动了软件工程界，使之重视并认真对待过程改善工作。软件 CMM 之后出现了一系列的能力成熟度模型，包括人的能力成熟度模型（P-CMM）（Curtis 等，2001），以及系统工程能力模型（Bate，1995）。

其他机构也开发了能力成熟度模型。能力评估和过程改善的 SPICE 方法（Paulk 和 Konrad，1994）要比 CMM 水平更有柔性。它包括与 SEI 相似的成熟度水平，不同的是它还识别打破这些水平界限的一些过程，如客户 - 供应商过程。当成熟度水平在增长时，这些横切过程的性能必须

也得到改善。

20 世纪 90 年代的 Bootstrap 项目的目标是扩展和修改 SEI 成熟度模型，使之适应更多的公司。Bootstrap 模型（Haase 等，1994；Kuvaa 等，1994）使用 SEI 成熟度水平（见 26.5.1 节），它也提出了一个基础过程模型（基于欧洲航天局使用的模型），作为局部过程定义的起点。它也包括一个开发全公司范围的质量系统的准则以支持过程改善。

在试图集成已经开发的过多的基于过程成熟度概念的能力模型（包括自己的模型）中，SEI 着手进行一个新的关于开发一个集成的能力模型（CMMI）的项目。CMMI 框架代替了软件和系统工程的 CMM，并集成了其他的能力成熟度模型。它有两个实例，一个是阶段性的，一个是连续性的，并且解决某些在软件 CMM 中报告出来的弱点。

CMMI 模型（Ahern 等，2001；Chrissis 等，2007）目标是成为过程改善的框架，能广泛适用于多种企业。它的阶段性版本是与软件 CMM 一致的，并允许评估机构的系统开发和管理过程且赋予从 1~5 的成熟度水平。它的连续版本允许过程成熟度的细粒度的分类。此模型提供一种方法将 22 个过程域（见图 26-7）分为从 0~5 六个等级。

CMMI 模型十分复杂（多于 1000 页的描述），这里将它做了重大简化以方便讨论。主要模型组件如下：

1. 一组过程域。过程域和软件过程活动有关。CMMI 识别 22 个与软件过程能力和改善相关的过程域。它们在连续的 CMMI 模型中被分成 4 组。这些组和相关的过程域在图 26-7 中列出。

2. 一些目标。目标是对一个机构要达到的期望状态的抽象描述。CMMI 有与每个过程域相关的特定目标，并定义了每个域的期望状态。它也定义了将把好的实践制度化相关联的通用目标。图 26-8 给出了 CMMI 的特定和通用的目标的例子。

3. 一组好的实践。实践是对达到目标的方法的描述。在过程域中与每个目标相关联的有几个特定的和通用的实践。建议的实践的例子在图 26-9 中给出。然而，CMMI 认识到，目标比达到目标的方法更重要。机构可以使用任何可以达到 CMMI 的目标的实践途径，他们不必采用 CMMI 的建议实践。

范 畴	过 程 域
过程管理	机构过程定义 (OPD) 机构过程焦点 (OPF) 机构培训 (OT) 机构过程性能 (OPP) 机构创新和部署 (OID)
项目管理	项目规划 (PP) 项目监控 (PMC) 供应商协议管理 (SAM) 集成项目管理 (IPM) 风险管理 (RSKM) 量化项目管理 (QPM)
工程	需求管理 (REQM) 需求开发 (RD) 技术解决方案 (TS) 产品集成 (PI) 检验 (VER) 有效性验证 (VAL)
支持	配置管理 (CM) 过程 and 产品质量保证 (PPQA) 度量和分析 (MA) 决策分析和解决方案 (DAR) 原因分析和解决方案 (CAR)

图 26-7 CMMI 中的过程域

目 标	过 程 域
当项目执行或者结果严重偏离计划的时候纠正行动设法停止	项目监控中的特殊目标
项目的实际执行和进程根据项目计划得到监控	项目监控中的特殊目标
需求得到分析和有效性验证，所需要的功能的定义得以完成	需求开发中的特殊目标
缺陷和其他问题的根本原因得到系统地确定	原因分析和解决方案中的特殊目标
过程被制度化为已定义的过程	普通目标

图 26-8 CMMI 中特定和通用目标实例

目 标	相 关 实 践
需求得到分析和有效性验证, 所需的功能得到定义	系统地分析导出的需求, 以确保它们是必要和充分的
	利用多种适当技术验证需求以确保得到的产品能够在用户环境中按照用户所期待的方式执行
系统地确定缺陷和其他问题的根本原因	为分析阶段选择关键缺陷和其他问题
	对所选择的缺陷和其他问题执行原因分析并提出解决它们的行动
将过程制度化为一个已定义的过程	建立和维持对规划和执行需求开发过程的机构策略
	赋予关于下面这些方面的责任和权力, 即执行过程、开发工作产品、为需求开发过程提供服务

图 26-9 CMMI 中目标和相关的实践

通用目标和实践不是技术性的而是与好的实践制度化相关联的。因此, 它意味着什么是依赖于机构的成熟度的。在成熟度发展的早期阶段, 制度化可能意味着保证计划的建立和过程的定义面向公司中所有的软件开发。然而, 对于一个拥有更成熟的和先进过程的机构, 制度化可能意味着在整个机构中使用统计的和其他的量化技术引入过程控制。

CMMI 评估是要对机构中的过程进行检查, 并对这些过程或过程域给出评级。对每个过程域的成熟度水平采用 6 级尺度进行度量。他们的观点是过程越成熟, 过程就越优良。这个 6 级过程域的水平指派是:

- 1. **不完整** 与过程域关联的至少一个特定目标没有得到满足。这个水平上没有通用目标, 因为对一个不完整的过程的制度化是没有意义的。
- 2. **已执行** 与过程域关联的目标都已经满足, 对所有的过程、所执行的工作范围显式地设定出来并和团队成员沟通。
- 3. **已管理** 在这个水平上, 与过程域关联的目标都能达到, 定义每个过程应该在什么时间使用的机构策略是到位的。这里必须有文档化的项目计划, 以定义项目目标。资源管理和过程监控程序必须要在整个机构中就绪。
- 4. **已定义** 在这个水平上, 焦点是对过程的机构标准化和部署。每个项目都有一个受到管理的过程, 此过程是对已定义的机构过程集合中的过程根据项目需求调整得到的。必须收集过程资产和过程度量, 并将之用于将来的过程改善中。
- 5. **已量化管理** 在这个水平上, 机构要负责使用统计和其他量化方法来控制子过程。即收集到的过程和产品度量必须使用到过程管理中来。
- 6. **优化** 在这个最高水平上, 机构必须使用过程和产品度量来驱动过程改善。必须要对趋势加以分析并且对过程加以调整来适应变更的业务需要。

这里是对能力水平的一种非常简化的描述, 如果要运用到实际中, 还需要做出更多细节描述。这些水平是递进的, 从最低层的显式的过程描述, 经过过程的标准化的, 再到最高层的由过程度量和软件驱动的过程变更和改善。为改善其过程, 公司应该致力于对与其业务相关的过程群提高成熟度水平。

26.5.1 分阶段的 CMMI 模型

分阶段的 CMMI 模型和软件能力成熟度模型 (CMM) 模型类似, 它提供了一种方法, 评估

机构过程能力并分为5个等级。它规定了在每一级中必须要达到的目标。通过在每一级上实现的工作，从模型的较低的级别向较高级别的移动，逐步达到过程改善。

分阶段 CMMI 模型的五级如图 26-10 所示。它们对应连续模型 1~5 的能力水平。分阶段的 CMMI 模型和连续 CMMI 模型的主要区别在于，分阶段模型用于评估机构整体能力，而连续模型度量的是机构内部的特别过程域的成熟度。

每个成熟度水平都有相关的一组过程域和通用目标。这些反映了优良的软件工程和管理实践以及过程改善的制度化。较低的成熟度水平可以通过引入好的实践达到，但是，更高的水平就需要进行过程度量 and 改善方可达到。

例如，与第二级（已管理级）相关的模型中所定义的过程域是：

1. **需求管理** 管理项目的产品和产品组件的需求，并且发现在这些需求和项目规划及工作产品之间的一致性。
2. **项目规划** 建立和维护定义项目活动的计划。
3. **项目监控** 提供对项目进展的了解，这样当项目较之规划明显有偏差时能采取适当的纠正行动。
4. **供应商协议管理** 管理来自项目外部供应商的产品和服务，这些是有正式协议的。
5. **度量和分析** 开发和维持用于支持对管理信息的需要的度量能力。
6. **过程 and 产品质量保证** 为员工和管理层提供对过程及相关工作产品的一个客观洞察力。
7. **配置管理** 使用配置识别、配置控制、配置状况记账和配置审计来建立和维护工作产品的完整性。

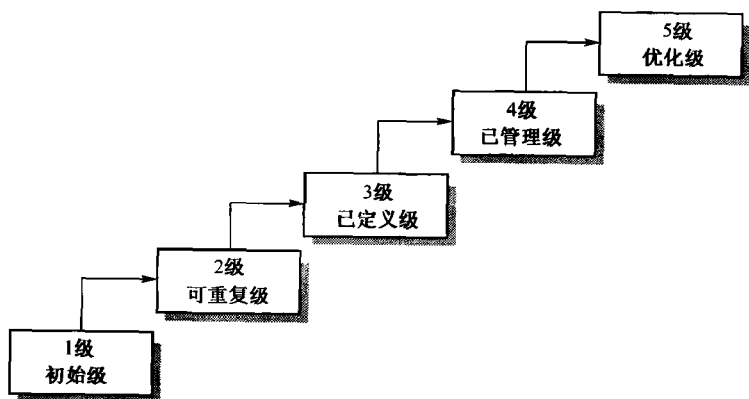


图 26-10 CMMI 阶段成熟度模型

除了这些特定的实践，在 CMMI 模型的第二级上运作的机构应该已经达到了将每个过程制度化为已管理过程这样的通用目标。机构实践的例子，与项目规划相关的能将项目规划过程变成已管理过程的有：

- 为规划和执行项目规划过程建立和维护机构策略。
- 为执行项目管理过程、研制工作产品和对过程提供服务等提供充足的资源。
- 按照计划监控项目规划过程，并采取适当的纠正行动。
- 与高层管理一道对项目规划过程的活动、状况和结果进行复查并解决出现的问题。

分阶段 CMMI 的优势是它与 20 世纪 80 年代提出的软件成熟度模型兼容。许多公司明白这个模型并且致力于使用这个过程改善模型。因此他们能直接从成熟度模型转移到分阶段 CMMI 模型。另外，分阶段模型为机构定义了明确的改善途径。它们能从第二级到第三级，如此等等。

然而，分阶段模型（以及软件 CMM）的缺点是它的指定特性。每个成熟度水平有其自己的目标 and 实践。分阶段模型假设在转移到另一个水平之前当水平的所有的目标和行为都已经实现。然而，机构的环境可能是这样：更恰当的做法是在低水平实践完成之前去实现更高级水平的目标和实践。如果机构这样做，成熟度评估会对它的能力给出错误印象。

26.5.2 连续 CMMI 模型

连续成熟度模型不根据离散等级来分类机构。而是一种考虑单个或一组实践以及评估每个过程组中好的实践的使用的一种细粒度模型。因此，成熟度评估不是一个单个的值，而是一组值，能为每个过程或一组过程给出机构的成熟度。

连续 CMMI 模型对过程域的分级如图 26-7 所示，为每个过程域赋值一个 0 ~ 5 级的能力评价（如先前介绍的）。通常，机构对不同的过程域在不同的成熟度水平上运作。因此，连续 CMMI 评估的结果是一个能力概况，表现每个过程域和与他关联的能力评估。图 26-11 给出了在不同能力水平上过程的能力概况的一个片段。例如，它说明了配置管理中成熟度的水平高，但风险管理的成熟度很低。公司可能研制实际的能力概况和目标的能力概况，目标概况反映的是它们希望达到的过程域的能力级。

连续模型的主要优点是公司可以根据他们的愿望和需求来挑选和最终决定要改善的过程。以作者的经验，不同的机构类型有不同的过程改善需求。例如，开发航空软件的公司可能关注系统描述、配置管理和有效性验证，而 Web 软件开发公司可能更多的是关心面向客户的过程。阶段性模型要求公司轮流关注不同的阶段。相反，连续 CMMI 允许自主性和灵活性，仍然允许公司在 CMMI 改善框架中运行。

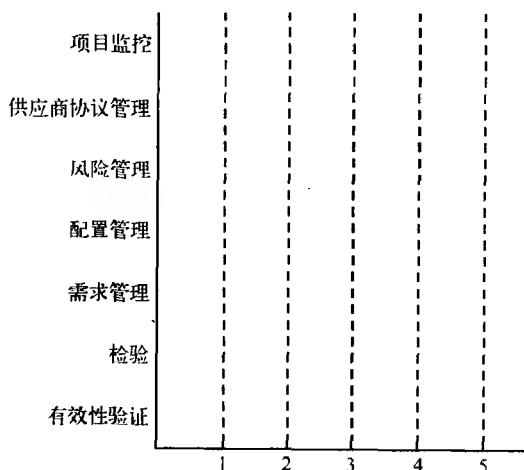


图 26-11 过程能力概况

要点

- 过程改善的目标是更高的产品质量，降低的过程开销，以及更快的软件交付速度。
- 过程改善的主要方法有：目标是减少过程成本的敏捷方法；基于更好的过程管理和好的软件工程实践的使用的成熟度的方法。
- 过程改善循环包括过程度量、过程分析、建模和过程变更。
- 过程模型，展现了过程中的活动以及它们和软件产品的关系，用于过程描述。但是在实际中，参与软件开发的工程人员总是调整模型使其适应他们的具体环境。
- 应该用度量来对所使用的软件过程的特殊问题做出回答。这些问题应该基于机构的改善目标。
- 在度量过程中使用的 3 类过程量度是：时间量度，资源使用量度，以及事件量度。
- CMMI 成熟度模型是一种综合的过程改善模型，既支持阶段性过程改善也支持连续性过程改善。
- CMMI 模型中的过程改善是基于达到一组关于好的软件工程实践的目标，并且为达到这些目标去描述、标准化和控制所采用的实践。CMMI 模型包括建议使用的实践，但并不强制使用。

进一步阅读材料

《Can you trust software capability evaluations》这是一篇对能力评估主题持怀疑态度的文章，评估了公司的过程成熟度，并讨论为什么评估不会给出一个机构成熟度的正确反映 (E. O'Connell and H. Saiedian, IEEE Computer, 33 (2), February 2000)。http://dx.doi.org/10.1109/2.820036。

《Software Process Improvement: Results and Experience from the field》这本书收集了几个小型和中型挪威公司的过程改善的案例研究。同时也包括了对过程改善的一般问题的很好的介绍 (Conradi, R., Dybaa, T., Sjøberg, D., Ulsund, t. (eds.), Springer, 2006)。

《CMMI: Guidelines for Process Integration and Product Improvement, 2nd edition》CMMI 的综述。CMMI 很庞大而且复杂，实际可能不容易读和理解。这本书通过加入一些轶事和历史材料，使其更易阅读。但有时候仍然很难理解 (M. B. Chrissis, M. Konrad, S. Shrum, Addison - Wesley, 2007)。

练习

- 26.1 软件过程改善的敏捷方法和过程成熟度方法的关键不同是什么？
- 26.2 在什么情况下产品质量可能决定于开发团队的质量？举例说明什么类型的软件产品特别依赖于个人的天赋和能力。
- 28.3 试列举 3 个为支持机构过程改善项目而开发的专门软件工具。
- 26.4 假设机构的过程改善目标是增加可复用组件的数量，这些组件是在开发过程中自行生产出来的。指出在上述情况下 GQM 范式可能存在的 3 个问题。
- 26.5 描述 3 种类型的软件过程量度，这些量度可以收集用来作为过程改善的一部分。给出每一种类型量度的例子。
- 26.6 为评估和排序过程变更建议设计一个过程。将这个过程用一个过程模型记录下来，该模型能说明在这个过程中的角色。必须使用 UML 活动图或者 BPMN 描述过程。
- 26.7 给出过程改善框架（如 CMMI 内所使用的）中嵌入的过程评估和改善方法的两个优点和两个缺点。
- 26.8 在什么情况下你会建议使用 CMMI 的阶段性表示？
- 26.9 使用过程成熟度模型，专注于要达到的目标而不是引入的好的实践，这有什么优点和缺点。
- 26.10 若过程改善程序包括度量人在过程中的工作和引入变更到过程中，那么它是否有内在的不人道性？对过程改善程序会发生哪些抵触行为？为什么？

参考书目

- Ahern, D. M., Clouse, A. and Turner, R. (2001). *CMMI Distilled*. Reading, Mass.: Addison-Wesley.
- Basili, V. and Green, S. (1993). 'Software Process Improvement at the SEL'. *IEEE Software*, 11 (4), 58-66.
- Basili, V. R. and Rombach, H. D. (1988). 'The TAME Project: Towards Improvement-Oriented Software Environments'. *IEEE Trans. on Software Eng.*, 14 (6), 758-773.
- Bate, R. 1995. 'A Systems Engineering Capability Maturity Model Version 1.1'. Software Engineering Institute.
- Chrissis, M. B., Konrad, M. and Shrum, S. (2007). *CMMI: Guidelines for Process Integration and Product Improvement, 2nd edition*. Boston: Addison-Wesley.
- Curtis, B., Hefley, W. E. and Miller, S. A. (2001). *The People Capability Model: Guidelines for Improving the Workforce*. Boston: Addison-Wesley.

- Haase, V., Messnarz, R., Koch, G., Kugler, H. J. and Decrinis, P. (1994). 'Bootstrap: Fine Tuning Process Assessment'. *IEEE Software*, **11** (4), 25–35.
- Hammer, M. (1990). 'Reengineering Work: Don't Automate, Obliterate'. *Harvard Business Review*, July–August 1990, 104–112.
- Humphrey, W. (1989). *Managing the Software Process*. Reading, Mass.: Addison-Wesley.
- Humphrey, W. S. (1988). 'Characterizing the Software Process'. *IEEE Software*, **5** (2), 73–79.
- Humphrey, W. S. (1995). *A Discipline for Software Engineering*. Reading, Mass.: Addison-Wesley.
- Kuvaja, P., Similä, J., Krzanik, L., Bicego, A., Saukkonen, S. and Koch, G. (1994). *Software Process Assessment and Improvement: The BOOTSTRAP Approach*. Oxford: Blackwell Publishers.
- Osterweil, L. (1987). 'Software Processes are Software Too'. *9th Int. Conf. on Software Engineering*, IEEE Press, 2–12.
- Ould, M. A. (1995). *Business Processes: Modeling and Analysis for Re-engineering and Improvement*. Chichester: John Wiley & Sons.
- Paulk, M. C., Curtis, B., Chrissis, M. B. and Weber, C. V. (1993). 'Capability Maturity Model, Version 1.1'. *IEEE Software*, **10** (4), 18–27.
- Paulk, M. C. and Konrad, M. (1994). 'An Overview of ISO's SPICE Project'. *IEEE Computer*, **27** (4), 68–70.
- Paulk, M. C., Weber, C. V., Curtis, B. and Chrissis, M. B. (1995). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Mass.: Addison-Wesley.
- Pulford, K., Kuntzmann-Combelles, A. and Shirlaw, S. (1996). *A Quantitative Approach to Software Management*. Wokingham: Addison-Wesley.
- Sommerville, I. and Sawyer, P. (1997). *Requirements Engineering: A Good Practice Guide*. Chichester: John Wiley & Sons.
- White, S. A. 2004. 'An Introduction to BPMN'. <http://www.bpmn.org/Documents/Introduction%20to%20BPMN>.

术 语 表

abstract data type (抽象数据类型)。一种由操作而不是表示定义的类型。其表示是私有的,只能由定义它的操作访问。

activity (PERT) chart (活动图)。项目管理者用来显示完成的任务间的依赖关系的图。图显示了任务、预期完成的时间和任务间的依赖关系。关键路径是在活动图中最长的路径(根据完成任务需要的时间)。关键路径定义了完成项目需要的最短时间。

Ada (Ada 语言)。一种程序语言,是 20 世纪 80 年代为美国国防部开发军用软件而开发的作为一种标准的语言。它基于对 20 世纪 70 年代的程序语言的研究,包括类似于抽象数据类型和并发性的概念。它仍然应用于大型复杂的军事和航天系统。

agile manifesto (敏捷宣言)。一组原理,封装了敏捷软件开发方法背后的思想。

agile methods (敏捷方法)。适合于快速软件交付的软件开发方法。软件以增量式开发和交付,过程文档和管理手续减到最小。开发集中在代码本身上,而不是那些支持文档上。

algorithmic cost modelling (算法成本建模)。一种软件成本估算的方法,由公式估算项目成本。公式中的参数是项目和软件本身的属性。

application family (应用族)。一组软件应用程序,有共有的体系结构和通用的功能。这些应用能够通过修改组件和程序参数来适应特殊客户的需求。

application framework (应用框架)。针对某些特殊领域的一种通用结构,构成一系列应用的基础。应用框架通常实现为一组具体的和抽象的类,这些类通过特化和实例化来创建一个应用。

Application Program Interface (API) (应用程序接口)。一种接口,通常被指定为一组操作,这些操作由应用程序定义,允许访问到程序提供的功能。这就意味着其他程序能够直接调用此功能,而不仅仅是通过用户接口访问这些功能。

architectural pattern (style) (体系结构模式(风格))。软件体系结构的一种抽象描述,这是经过无数次不同软件系统的尝试和测试过的。模式描述包括使用模式最适合的场合以及体系结构中的组件组成的信息。

aspect-oriented software development (面向方面的软件开发)。一种软件开发的方法,它结合生产式和基于组件的开发方法。找出程序中的横切关注点,这些关注点的实现被定义为方面。方面包括它们要在何处整合入程序的信息,然后程序编织器将这些方面编织到程序中合适的地方。

aspect weaver (方面编织器)。这是一个程序,总是作为编译系统的一部分存在。编译系统处理面向方面的程序并修改代码以在指定的程序位置包含进已定义的方面。

availability (可用性)。在有请求的时候系统能提供服务。可用性通常以小数来表示,因此 0.999 的可用性的意义是系统能够在 1000 次请求中成功提供 999 次服务。

bar chart (条形图)。项目管理者用来显示项目任务、任务关联的进度和影响这两项指标的人员的图。图中显示了任务开始和结束的日期还有紧卡时间线的人员分配。

BEA。是美国的一个 ERP 系统的卖主。

black-box testing (黑盒测试)。一种测试方法,测试人员不访问系统或其组件的源代码。测试从系统描述中导出。

BPMN。Business Process Modeling Notation。定义工作流的符号系统。

brownfield software development。在多个现有系统环境中的软件开发。这些现有系统必须与待开发的系统集成在一起。

C (C 语言)。一种程序语言，最初是开发用来帮助实现 Unix 系统的。C 语言是一种相对低级的系统实现语言，允许访问系统硬件，并且可以被编译为高效代码。C 语言广泛应用于低级系统编程和嵌入式系统开发。

C++ (C++ 语言)。一种面向对象的程序语言，是 C 语言的超集。

C#。一种面向对象的编程语言。由微软公司所开发。它与 C++ 有很多共同之处，但是又包含了允许更多的编译时类型检查的特征。

CASE (Computer-Aided Software Engineering, 计算机辅助软件工程)。使用自动化支持的软件开发过程。

CASE tool (CASE 工具)。一种软件工具，例如设计编辑器或程序调试器，用来支持软件开发过程中的活动。

CASE workbench (CASE 工作台)。一个集成的 CASE 工具集，一起工作来支持主要过程活动（比如，软件设计或者配置管理等）。

change management (变更管理)。这是一个对提议的软件系统进行变更的记录、检查、分析、估计和实现的过程。

class diagram (类图)。这是 UML 的一种图的类型，描述系统中对象类以及它们之间的关系。

client-server architecture (客户 - 服务器体系结构)。一种分布式系统的体系结构模型，系统功能通过服务器提供的一组服务实现。使用服务的客户计算机可以访问这些功能。这种方法的几种变形使用多个服务器，例如三层的客户 - 服务器体系结构。

cleanroom software engineering (净室软件工程)。软件开发的一种方法，其目标是避免将错误导入软件（类似于半导体制作中用的净室）。这个过程包括形式化的软件描述，从描述向程序的结构化转换，开发程序正确性证明和统计程序测试。

cloud computing (云计算)。这是一种提供计算和应用服务的方式，通过因特网使用由外部提供商所提供的云一样多的服务器集群。“云”的实现是使用大量的商用计算机和虚拟技术，以便有效地使用这些系统。

CMM。软件工程研究所 (The Software Engineering Institute) 的能力成熟度模型 (Capability Maturity Model)。它用于评估机构的软件开发成熟度水平。目前已经被 CMMI 所代替，但还是得到广泛的使用。

CMMI (CMMI)。一种集成的过程能力成熟度建模方法，采用了良好的软件工程实践和集成质量管理。它支持离散的和连续的成熟度建模，并且集成了系统和软件工程过程成熟度模型。

code of ethics and professional practice (道德规范和职业准则)。一系列指导方针，它规定了软件工程师应有的道德规范和职业行为。由主要的美国专业协会 [美国计算机学会 (ACM) 和电气和电子工程师协会 (IEEE)] 制定，制定了 8 个方面的道德规范行为，包括：公众、客户和雇主、产品、评价、管理、同事、职业和自身。

COM+ (COM+)。为微软平台而设计的组件模型和辅助中间件，已被 .NET 代替。

Common Request Broker Architecture (CORBA, 通用对象请求代理体系结构)。由对象管理组织 (OMG) 提议的一系列标准，定义了一种分布对象模型和对象通信，在分布式系统开发中有影响力，但现在已很少使用。

component (组件)。软件中可部署的独立的单元，完全由一组接口定义和访问。

component model (组件模型)。一系列关于组件实现、文档化和部署的标准。覆盖了可能

由组件提供的特殊接口、组件命名、组件互操作和组件合成。组件模型提供了支持组件执行的中间件的基础。

Component-Based Software Engineering (CBSE, 基于组件的软件工程)。通过独立、可部署的组件 (与组件模型并用) 集成来开发软件的方法。

configuration item (配置项)。计算机可读的单元, 比如一个文档或一个源代码文件, 为变更提供服务。这里的变更必须得到配置管理系统的控制。

configuration management (配置管理)。管理进化中软件产品的变更的过程。配置管理包括配置规划、版本管理、系统构建和变更管理。

COnstructive COst MOdelling (COCOMO, 构造性成本建模。这是一组算法成本估算模型, 最先提出于 20 世纪 80 年代早期, 此后几经修改和更新以反映新的技术和变化了的软件工程实践。

CORBA component model (CORBA 组件模型)。为用在 CORBA 平台上而设计的组件模型。

control metric (控制量度)。一种软件量度, 允许管理者基于软件过程的信息或者是基于软件产品的信息做出规划决策。绝大多数控制量度是过程量度。

critical system (要求极高的系统)。指一旦失败了会造成经济、人员或环境上的巨大损失的计算机系统。

CVS。一种广泛使用的开放源码的软件工具, 用于版本管理。

data processing system (数据处理系统)。是一个用来处理大量的结构化数据的系统。这些系统通常按照输入 - 处理 - 输出的模式处理批量数据。账单和发票系统以及付款系统是数据处理系统的实例。

denial of service attack (拒绝服务攻击)。对基于 Web 的软件系统的一种攻击, 试图使系统过载使之不能提供正常的服务给用户。

dependability (可依赖性)。系统的可依赖性是一个综合考虑系统的安全性、可靠性、可用性、信息安全性和其他属性的总的属性。系统的可依赖性反映了用户可以信赖系统的程度。

dependability case (可依赖性论据)。是一种结构化文档, 用来备份系统开发者关于系统可依赖性的声明。

dependability requirement (可依赖性需求)。一种有助于达到所要求的系统可依赖性的系统需求。非功能性的可依赖性需求指定了可依赖性属性值; 功能性可依赖性需求是指避免系统缺陷和失败、检测, 容错或从系统缺陷和失败中恢复的功能性需求。

design pattern (设计模式)。一种实验证明效果良好的解决某个一般问题的方法, 它捕捉经验和好的实践, 以一种可以复用的形式表示, 是一种能以很多种方法实例化的抽象描述。

distributed system (分布式系统)。一种软件系统, 其软件子系统或组件在不同的处理器上执行。

domain (领域)。软件所拥有的特定问题或所使用的业务范围。领域的例子包括: 实时控制、业务数据处理、电信交换。

domain model (领域模型)。领域抽象的定义, 如政策、规程、对象、关系、事件。它可以作为某些问题领域的知识基础。

DSDM。这是 Dynamic System Development Method 的缩写。据说是首个敏捷开发方法之一。

embedded system (嵌入式系统)。嵌入在硬件设备当中的软件系统 (例如, 在手机里的软件系统)。嵌入式系统总是实时系统, 所以必须对环境中的事件做出及时的响应。

emergent property (总体特性)。只有当系统的所有组件集成在一起创建系统时才会显现的一种属性。

Enterprise Java Beans (EJB, 企业 Java beans)。一种基于 Java 的组件模型。

Enterprise Resource Planning (ERP) system (企业资源规划系统)。这是一种大型软件系统, 具有广泛的支持工商企业运行的能力, 并提供在各种能力间共享信息的手段。例如, 一个 ERP 系统可能包括对供应链管理、制造和分发的支持。对每个使用系统的单位需要根据它们的需求进行相应配置。

ethnography (深入实际)。可以用于引出需求和分析的一种观察技术。深入实际是要沉浸在用户环境中, 且能观察用户每天工作的习惯。软件支持的需求可以由这些观察中推导出来。

event-based systems (基于事件的系统)。是这样一种系统, 其操作控制取决于系统环境中产生的事件。绝大多数实时系统是基于事件的系统。

Extreme Programming (XP, 极限编程)。是一种敏捷的软件开发方法, 包括很多实践, 如基于情景的需求, 测试优先的开发和配对编程。

fault avoidance (缺陷避免)。以这种方式开发软件, 就不会向软件中引入缺陷。

fault detection (缺陷检测)。使用过程和运行时检查在导致系统失败前检测和除去缺陷。

fault tolerance (容错)。在执行过程中即使缺陷发生, 系统仍能继续工作的能力。

formal methods (形式化方法)。是一种软件开发的方法, 通过使用形式化数学结构如谓词和集合对软件建模。严格的变换将模型转换为代码。绝大多数情形是在要求极高的系统的描述和开发中使用它。

Gantt chart (甘特图)。条形图的另一个名称。

incremental development (增量式开发)。软件开发的一种方法, 软件的交付和部署以增量的形式进行。

information hiding (信息隐藏)。利用编程语言结构隐藏数据结构的表示并控制对这些结构的外部访问。

inspection (审查)。参见 program inspection。

insulin pump (胰岛素泵)。一种软件控制的医疗设备, 对糖尿病患者注射控制剂量的胰岛素。在本书中作为案例。

interface (接口)。对软件组件所关联的属性和操作的描述。接口用作访问组件功能的手段。

ISO 9000 (ISO 9000 标准)。是由国际标准化组织制定的一套质量管理过程的标准。ISO 9001 是非常适用于软件开发的 ISO 标准。这些标准也能用于组织内的质量管理过程保证。

iterative development (迭代式开发)。软件开发的一种方式, 软件描述、设计、编码和测试是交替进行的。

J2EE。是 Java 2 Platform Enterprise Edition 的缩写。这是一个复杂的中间件系统, 用于支持用 Java 所做的基于组件的 Web 应用开发。它包括用于 Java 组件、API 以及服务的组件模型。

Java (Java 语言)。一种面向对象的编程语言, 是由 SUN 公司设计的, 目的是使软件具有平台独立性。

language processing system (语言处理系统)。一个把一种语言翻译解释为另一种语言的系统。例如, 编译器就是一个语言处理系统, 它把程序源代码编译成目标代码。

legacy system (遗留系统)。一个社会技术系统, 它对某个机构是有用的或必需的, 但却是用过时的技术或方法开发的。由于遗留系统通常完成关键的业务功能, 不得不对它们进行维护。

Lehman's laws (Lehman 定律)。对影响复杂软件系统进化的因素的一组假设。

Maintenance (维护)。系统投入运行后, 对其进行变更的过程。

make。最早的系统构建工具之一, 广泛使用于 Unix/Linux 系统中。

Mean Time To Failure (MTTF, 平均失败时间)。在可靠性描述中使用, 指已发现的系统失

败的平均时间。

MHC-PMS。心理健康护理病人管理系统的缩写。在本书中作为案例。

middleware (中间件)。分布式系统中的基础软件,它负责管理系统中分布式实体和系统数据库的交互。中间件的例子有对象请求代理和事务管理系统。

model checking (模型检查)。一种静态检验方法,系统的状态模型得到全面分析以便发现不可到达的状态。

Model-Driven Architecture (MDA, 模型驱动体系结构)。基于构建一组系统模型的软件开发方法。它可以自动地或半自动化地执行并生成一个可执行的系统。

Model-Driven Development (MDD, 模型驱动的开发)。围绕 UML 表达的系统模型的软件工程方法,这里采用 UML 描述的模型,有别于用编程语言代码描述的任何模型。它拓展了 MDA,考虑除了开发之外的活动,例如需求工程和测试。

.NET。一种规模非常大的用于开发微软 Windows 应用的框架。包括:定义 Windows 系统中组件的标准,相关的支持组件执行的中间件的组件模型。

object class (对象类)。对象类定义对象的属性和操作方法。对象的创建是在运行时通过实例化类定义完成的。在一些面向对象的语言中,对象类名可以用作类型名。

Object Constraint Language (OCL, 对象约束语言)。作为 UML 一部分的一种语言,用于定义谓词,这些谓词应用于 UML 模型中的对象类和交互。使用 OCL 来定义组件是模型驱动开发的一部分。

Object Management Group (OMG, 对象管理组 (COMG))。由许多公司组成的组织,目的是制定面向对象开发的标准。由 OMG 提出的标准有 CORBA、UML 和 MDA 等。

object model (对象模型)。软件系统的一种模型,在这种模型中系统是由一组对象类和这些类之间的关系构成的。存在对象模型的多种不同的观点,例如状态观点和序列观点。

Object-Oriented (OO) development (面向对象的开发)。软件开发的一种方法,系统中的基本抽象是独立的对象。在描述、设计和开发中使用相同类型的抽象。

open source (开源)。一种软件开发方法,系统源代码公开,鼓励外部用户参与到系统的开发当中来。

pair programming (结对编程)。这是一种开发情形,程序员两两配对工作而不是各干各的。它是极限编程的基本部分。

peer-to-peer system (对等系统)。一种分布式系统,其中客户机和服务器没有明确的区分。系统中的计算机既可作为客户机,也可作为服务器。对等系统的应用包括:文件共享,即时消息,合作支持系统。

People Capability Maturity Model (P-CMM, 人员能力成熟度模型)。一种过程成熟度模型,反映一个机构如何能有效地管理机构中员工技能、培训和经验的模型。

predictor metric (预言者量度)。是一种软件量度,用于作为预测的基础。所谓的预测是预测软件系统的特性,包括可靠性或可维护性。

Probability Of Failure On Demand (POFOD, 请求失败的概率)。是一种可靠性量度,基于对软件系统在要求得到服务时失败的可能性的量度。

process improvement (过程改善)。一种对过程进行变更的过程,目的是使过程更加可预测或提高它的输出质量。例如,如果你的目标是减少交付软件的缺陷数目,你可以通过增加新的有效性验证活动来改善过程。

process maturity model (过程成熟度模型)。关于过程适应于过程改善所包括的好的实践、反映和度量能力的程度。

process model (过程模型)。是过程的抽象表示,过程模型可能来自于不同的观点,可以展示一个过程中涉及的活动、过程的产物、对过程所施加的约束和在过程中人员扮演的角色。

program evolution dynamics (程序进化的动态特性)。对进化中软件系统变更方法的研究。据称 Lehman 律控制着程序进化的动态特性。

program generator (程序生成器)。能从高层抽象描述产生另一个程序的程序,生成器可以嵌入知识,这些知识可以在每一次产生活动中重用。

program inspection (程序审查)。一种评审过程,是指有一组检查人员一行一行地审查程序,目的是发现程序中的错误。程序审查通常有一个常见编程错误清单。

Python。是一种编程语言,具有动态类型,特别适合于开发基于 Web 的系统。

Quality Assurance (QA, 质量保证)。定义怎样得到软件质量和怎样使得开发机构知道软件具有所要求的质量水平的总体过程。

quality plan (质量规划)。定义所需要使用的质量过程和步骤的计划。包括为产品和过程选择或实例化标准并定义最为重要的系统质量属性。

Rapid Application Development (RAD, 快速应用开发)。一种目的是快速交付软件的软件开发方法。它通常包括使用数据库编程和开发支持工具,如屏幕和报表生成器。

Rate Of Occurrence Of Failure (ROCOF, 失败出现率)。在某个给定时间段观察系统的失败次数得到的可靠性量度。

Rational Unified Process (RUP, Rational 统一过程)。一种通用软件过程模型,将软件开发表示为一个 4 个阶段的迭代活动,这 4 个阶段是开端、细化、构造、转换。开端阶段为系统建立一个业务案例,细化阶段定义体系结构,构造阶段实现系统,而转换阶段在客户环境中部署系统。

real-time system (实时系统)。一个必须对外界事件进行实时反应的系统。系统的正确性不只是依赖于它做什么同时依赖于它的反应速度。实时系统通常组织成一组协作的连续的进程。

reengineering (再工程)。修改软件系统使它更容易理解和改变。再工程通常包括:对软件和数据的重构建和组织,程序简化和文档化。

reengineering, business process (再工程,业务过程)。变更业务过程使之满足一些新的机构目标,例如降低成本和能更快速地执行。

reference architecture (参考体系结构)。一种通用系统体系结构,这是一种理想化的体系结构,包含系统可能包括的所有特征。这是一种使设计者了解此类系统一般结构的方法,而并非创建特定系统体系结构的基础。

release (发布版本)。软件系统的一个版本,对系统客户可用的版本。

reliability (可靠性)。系统提供所指定的服务的能力。可靠性可以定量地描述,如用请求失败概率 (POFOD) 或失败出现率描述。

reliability growth modeling (可靠性增长建模)。建立一个系统可靠性增长的模型,以反映系统在经过测试后或程序缺陷删除后可靠性的改变。

requirement, functional (需求,功能性的)。对系统中必须实现的一些功能或特征的一个声明。

requirement, non-functional (需求,非功能性的)。对施加于一个系统的约束或所期望的行为的声明。这个约束可能指的是正在开发的软件的整体特性或是开发过程。

requirement management (需求管理)。管理对需求变更的过程来保证变更是经过合理分析并在系统中全程跟踪的。

REST。REST 是 Representational State Transfer 的缩写。是一种基于简单客户机/服务器交互的

开发风格，它使用 HTTP 协议。REST 基于可识别的资源，即 URL 的理念。对资源的所有交互都是基于 HTTP POST、GET、PUT 和 DELETE 进行的。它目前已经广泛使用来实现低成本 Web 服务。

risk (风险)。一个不希望出现的输出会造成对取得某些目标的威胁。过程风险威胁着过程的进度或成本；产品风险意味着可能导致系统的一些需求不能达到。

risk management (风险管理)。识别风险、评估它们的严重性、规划措施以便在风险出现时能及时处理、监控软件和软件过程中的风险的一系列过程就是风险。

Ruby。一种程序设计语言，具有动态类型，尤其适合于 Web 应用的程序开发。

safety (安全性)。系统运行过程中避免灾难性的失败的能力。

safety case (安全用例)。一种结构化的论证说明系统是安全的或不安全。很多要求极高的系统必须具备相关的安全用例。这些安全用例是由外部监管者在核发使用许可前确定和认可的。

SAP。德国公司，开发了著名的广泛使用的 ERP 系统。它也指的是命名为 SAP 的 ERP 系统本身。

scenario (场景，情景脚本)。描述系统使用的典型方式或用户执行某些活动的典型方式。

Scrum。一种敏捷开发方法，基于冲刺短开发环。Scrum 可以用来作为敏捷项目管理的基础与其他敏捷方法（例如 XP）一起使用。

security (信息安全性)。系统保护自身免于偶然的和恶意的入侵的能力。信息安全性包括保密性、完整性和可用性。

SEI。是 Software Engineering Institute 的缩写。它是软件工程研究和技术的转移中心，创建的目的是改善美国公司的软件工程的标准。

sequence diagram (序列图)。描述为完成某些操作所需要的交互序列的图。在 UML 中，序列图可能与用例相关联。

server (服务器)。一个能对其他程序（客户机）提供服务的程序。

service。参见 Web service。

socio-technical system (社会技术系统)。是一种包含软硬件组件、对人的操作过程有良好定义的在组织内部运行的系统。因此，它受到机构的策略、规程及结构的影响。

software architecture (软件体系结构)。描述软件系统基本结构和组成的模型。

software life cycle (软件生命周期)。通常作为软件过程的另一个名字。最初提出这个术语指的是软件过程的瀑布模型。

software metric (软件量度)。软件系统或过程的属性，可以用数字形式表达和度量。过程量度是一个过程属性，例如完成任务的时间；产品量度是软件本身的属性，例如规模或复杂性。

software product line (软件产品线)。参看 application family。

software process (软件过程)。在软件系统的开发和进化过程中的活动及过程的相关集合。

spiral model (螺旋线模型)。开发过程的一种模型。这里过程被表示为一条螺旋线，螺旋线的每一圈包含过程的不同阶段。当从螺旋线的一个圈移动到另一个圈时，会重复过程的所有阶段。

state diagram (状态图)。一种 UML 图的类型，说明系统的状态以及触发系统从一个状态转变到另一个状态的事件。

static analysis (静态分析)。对程序源码进行基于工具的分析，以发现错误和异常。像对一个无中间使用的变量连续赋值这样的异常可能是编程错误。

structured method (结构化方法)。一种软件设计的方法。它定义需要开发的系统模型、应用于这些模型的规则和指导方针，以及在进行设计过程中应遵循的过程。

Structured Query Language (SQL, 结构化查询语言)。一种用于关系数据库编程的标准化

语言。

Subversion。一种广泛使用的开源系统构建工具，可在很多平台上使用。

system building（系统构建）。编译组成系统的组件或单元并将这些与其他组件相链接最终形成可执行程序的过程。系统构建通常是自动完成的，所以重编译可以得到最小化。这种自动化可能建立在语言处理系统内部（如 Java）或者是用软件工具来支持系统构建。

system engineering（系统工程）。定义系统、集成组件及测试系统是否符合其需求的过程。系统工程关注整个社会技术系统——软件、硬件及操作过程，而不仅仅是系统软件。

test coverage（测试覆盖）。测试整个系统代码的系统测试的有效性。某些公司有关于测试覆盖的标准（例如，系统测试应该保证所有程序语句至少要执行一遍）。

test-driven development（测试驱动的开发）。软件开发的一种方法，它是在写代码之前就写可执行测试。在对程序每做一次变更之后就自动运行一组测试。

transaction（事务）。一个与计算机系统交互的单元。事务是独立的和原子的（它们不能被分割成更小的单元），并且是恢复、一致性及并发性的基本单元。

transaction processing system（事务处理系统）。这是一种系统，它以保证事务处理过程不会相互影响，且单个事务的失败不会影响到其他事务及系统数据的方式工作。

Unified Modeling Language（UML，统一建模语言）。一种用于面向对象开发的图形语言，它包含多种系统模型提供系统的不同视图。UML 已经成为面向对象建模的事实上的标准。

use case（用例）。与系统交互的一种类型的描述。

user interface design（用户界面设计）。设计系统用户访问系统功能及显示系统所产生的信息的方式的过程。

validation（有效性验证）。检查系统是否满足了客户的需要和期望的过程。

verification（检验）。检查一个系统是否符合其描述的过程。

version management（版本管理）。对软件系统及其组件管理变更的过程，以便可能知道在每一个组件/系统版本中进行了哪些改变，也可以借此进行对组件/系统先前版本的恢复/重建。

waterfall model（瀑布模型）。一种软件过程模型，包含以下单独的开发阶段：描述、设计、实现、测试及维护。原则上，一个阶段必须在进入下一阶段之前完成。而实际上，在两个阶段之间存在迭代。

Web service（Web 服务）。可以通过因特网标准协议访问的独立软件组件，没有外部支持。基于 XML 标准的 SOAP（标准对象访问协议）用于网络服务信息交换。WSDL（Web 服务定义语言）用于定义 Web 服务界面。REST 方法或许也可以用于 Web 服务实现。

white-box testing（白盒测试）。程序测试的一种方法，测试是基于程序和组件的结构信息的。访问到源代码是白盒测试的关键所在。

wilderness weather system（野外气象系统）。一个收集偏远地区气象状况数据的系统。本书中作为用例。

workflow（工作流）。对业务过程的一种详细定义，目的是完成某项任务。工作流总是用图形的形式表达，给出单个过程活动及由每个活动所生产和消费的信息。

WSDL。一种基于 XML 的符号系统，用于定义 Web 服务的接口。

XML（可扩展标记语言）。可扩展标记语言。XML 是一种支持结构化数据交换的文本标记语言。每个数据域用标记符界定，标记符给出该数据域的信息。现今 XML 得到广泛应用并且成为 Web 服务协议的基础。

XP（极限编程）。极限编程（Extreme Programming）的常用缩写。

Z（Z 方法）。一种基于模型的、形式化描述语言，是由英国牛津大学开发的。